

# Visual C++ 游戏编程基础

Visual C++ Game  
Programming Basic Tutorial



肖永亮  
荣钦科技  
刘晓华  
飞思科技产品研发中心

丛书主编  
著  
改编  
监制



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>

随书光盘内容为  
书中实例源文件



本书介绍了如何以Visual C++及DirectX来设计游戏，主要是针对想学习游戏设计的初学者，通过本书深入浅出的概念讲解与实例相结合来逐步实现自己制作游戏的梦想。书中循序渐进地从游戏画面绘制、游戏动画技巧、游戏输入消息处理、游戏人工智能、游戏物理现象设计原理及如何进入3D世界等基本的游戏设计基础开始，到实际的程序范例编写，除了让初学者有清楚的基础概念以外，还能实际地应用于游戏设计，书中的最后一章以游戏项目开发为范例，完整地展示了初期规划及所有设计过程，随书光盘内容为书中范例源文件。

本书适合游戏开发人员及游戏相关专业师生学习使用。

# VISUAL C++ GAME PROGRAMMING BASE DESIGN



随书光盘内容为  
书中实例源文件



ISBN 7-121-00915-3



9 787121 009150 >

飞思在线: <http://www.fecit.com.cn>  
飞思科技产品研发中心总策划

本书贴有激光防  
伪标志，凡没有  
防伪标志者，属  
盗版图书



责任编辑: 王树伟

责任美编: 孙莹

ISBN 7-121-00915-3 定价: 39.00元  
(含光盘1张)

游戏学院经典书丛

# Visual C++ 游戏编程基础

Visual C++ Game  
Programming Basic Tutorial



肖永亮  
荣钦科技  
刘晓华  
飞思科技产品研发中心

丛书主编  
著  
改编  
监制

电子工业出版社  
Publishing House of Electronics Industry  
北京·BEIJING





# 内容简介

本书介绍了如何以 Visual C++ 及 DirectX 来设计游戏,主要针对想学习游戏设计的初学者,通过本书深入浅出的概念与实例相结合来逐步实现自己制作游戏的梦想。书中循序渐进地从游戏画面绘制、游戏动画技巧、游戏输入消息处理、游戏人工智能、游戏物理现象设计原理及如何进入 3D 世界等基本的游戏设计基础开始,到实际的程序范例编写,除了让初学者有清楚的基础概念以外,还能实际地应用于游戏设计,书中的最后一章以游戏项目开发为范例,完整地展示了初期规划及所有设计过程,随书光盘内容为书中范例源文件。

本书适合游戏开发人员及游戏相关专业师生学习使用。

本书繁体字版名为《Visual C++ 游戏设计魔法宝典》,由荣钦科技股份有限公司授权出版,著作权归荣钦科技股份有限公司所有。本书简体字中文版授权电子工业出版社出版。专有出版权属电子工业出版社所有,未经本书版权所有者和本书出版者书面许可,任何单位和个人不得以任何形式或任何手段复制或传播本书的部分或全部内容。

版权贸易合同登记号 图字:01-2005-0486

## 图书在版编目(CIP)数据

Visual C++ 游戏编程基础 / 荣钦科技著;刘晓华改编. —北京:电子工业出版社,2005.5

(游戏学院经典书丛/肖永亮主编)

ISBN 7-121-00915-3

I.V... II.①荣...②刘... III.①C 语言—程序设计②游戏—应用程序—程序设计 IV.①TP312②G899

中国版本图书馆 CIP 数据核字(2005)第 008517 号

责任编辑:王树伟

印刷:北京天宇星印刷厂

出版发行:电子工业出版社

北京海淀区万寿路 173 信箱 邮编:100036

经销:各地新华书店

开本:850×1168 1/16 印张:25.75 字数:700.4 千字

印次:2005 年 5 月第 1 次印刷

印数:6 000 册 定价:39.00 元(含光盘 1 张)

凡购买电子工业出版社的图书,如有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系电话:010-68279077。质量投诉请发邮件至 zlts@phei.com.cn,盗版侵权举报请发邮件至 dbqq@phei.com.cn。



## 荣钦科技

荣钦科技拥有一支专精各种计算机知识领域的专家团队。高凝聚力产生高含量的IT信息资讯图书。荣钦科技在游戏开发、网络开发、影像处理、硬件编程、程序语言等领域都有独到的见解和核心技术实力。更为重要的是，荣钦科技团队拥有一颗永不懈怠的上进心，这正是保证他们推出的每一种图书品质优秀的力量源泉。荣钦希望和读者的每一次交流都可以让彼此得到最大的收获。

VANGUARD BOOKS  
PROGRAMMING AND TECHNOLOGY



**肖永亮**博士，著名数字媒体专家，教授；任教美国纽约大学，北京师范大学；美国计算机协会图形图像学会 (ACM SIGGRAPH) 北京分会会长；北京师范大学艺术与传媒学院副院长，数字媒体研究所所长，博士生导师；清华大学文化产业中心兼职研究员；南昌大学客座教授；中华民族文化促进会理事；全国青联留学人员联谊会理事；新闻传媒专业委员会秘书长；中国旗美科技协会理事，新媒体学会会长。先后在美国路易威尔大学获高等教育管理文学硕士，哲学博士学位；肯塔基大学数学系，纽约西奈山医学院完成博士后研究；主要从事电脑算法理论、人工智能，软件开发应用，网络系统管理，计算机图像处理，三维动画游戏等研究，在国际一流杂志上发表过近十篇学术论文并载入美国科学与工程世界名人录。曾就职于美国新闻集团FOX 影视公司蓝天制片厂并任总工程师，执掌公司的技术发展方向和引领前沿技术，领导和协调研发部，网管部，系统工程部，软件编程部和生产流程部，设计和完成多项几千万美元的生产项目，参加多部居电影票房榜首的影片特技和轰动性电视广告的设计制作，其中其参与设计的动画片《邦尼》荣获第71届奥斯卡最佳动画短片金像奖。

飞思出品

Flying with the TEchnology.Thinking about the FuTure



## 编委会

主 编 肖永亮

委 员 (排名不分先后)

成 飞

美国好莱坞独立游戏设计师

狄亚铭

美国 Turbine 娱乐软件公司亚洲区艺术总监,“指环王”游戏产品设计师

耿卫东

浙江大学计算机学院教授、博士生导师,浙江大学 CAD&CG 国家重点实验室副主任

肖永亮

北师大艺术与传媒学院副院长、教授、博士生导师,北京 SIGGRAPH 会长

于金辉

浙江大学 CAD&CG 国家重点实验室研究员,博士生导师,浙江大学计算机学院数字媒体与网络技术系主任

王 敏

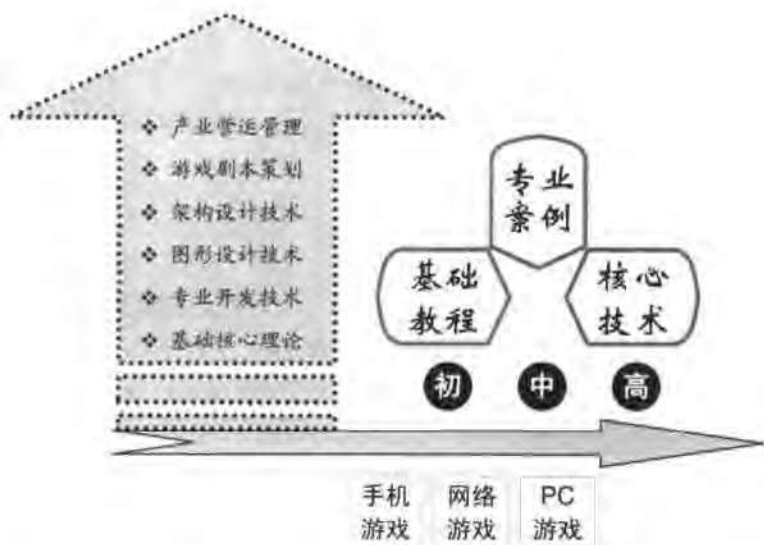
中央美院设计学院院长、教授

潘志庚

浙江大学计算机学院教授、博士生导师,浙江大学 CAD&CG 国家重点实验室研究员,中国图形图像学会常务理事、副秘书长

随着近年来游戏产业的持续升温，游戏产业的价值引起了国人的重视。但是由于国内缺乏正规的游戏教育，中国游戏产业的命脉被握在了韩国、日本、欧美厂商的手中。国产游戏产业也在夹缝中摸索生存的道路。要提高中国游戏开发的水平与质量，提升中国游戏在国际上的竞争力，大力加强游戏教育已是当务之急。目前国内已有一些院校开设了游戏教育专业，但与之相配套的教育知识体系还未成熟，众多游戏开发企业也在苦苦寻找能够进行游戏开发的“千里马”。

为了推动游戏教育的发展，引导和规范游戏开发的人才教育，培养具有专业水平的游戏开发人才，由电子工业出版社计算机研发部策划，飞思科技产品研发中心、飞思数码产品研发中心联合国内相关权威机构和多家培训机构合作，邀请众多知名院校专家和培训专家，结合院校教学需求、培训机构教学需求和社会读者自学需求，体现游戏领域的架构设计、编程开发、人机界面设计和行业运营等方面技术，整合各方面资源，打造国内优秀的、原创的“游戏学院经典书丛”系列。本套丛书规划如下：



1. 全套丛书涵盖手机游戏、网络游戏和 PC 游戏三个方向，涉及当前主流的三类游戏，提供三类游戏领域的特定知识内容；
2. 分别从“基础核心理论”、“专业开发技术”、“图形设计技术”、“架构设计技术”、“游戏剧本策划”、“产业运营管理”六个方向规划全套丛书，综合电脑游戏的每个核心节点上知识，满足行业各类人士的阅读需求；
3. 丛书在体系上呈现一种从“基础教程”、“核心技术”到“专业案例”的、由浅入深路线，满足初、中、高各层次读者需求。

丛书涉及领域广泛，纵深适宜。本套丛书涵盖了游戏产业领域的各个方向，同时从覆盖各个层次的读者，既有适合作为学校和培训机构的教材，也有适合读者自学的教辅。丛书主要特色如下：

- **目标学习，案例导航**

将游戏开发中涉及到的知识点以案例导航的形式进行介绍，使读者能够快速掌握要点，进入真正的开发状态。

- **作者专业，实践性强**

丛书作者有来自韩国的游戏开发专家，中国台湾的游戏开发团队、有多次获奖的国内外游戏美术设计师，也有来自国内著名高校中专门从事游戏研发的教授。高水平的作者确保了丛书的技术先进性和可实践性。

丛书在规划出版过程中，得到了业界培训机构、协会以及知名院校专家的大力支持，从而使本丛书的内在质量与外在品质都比同类图书更胜一筹，在此我们表示感谢。

我们临出版之残酷竞争而不惧，旌旗猎猎而异军突起，这与广大读者的支持是分不开的。为使我们的脚步更坚实、使我们的队伍永远保持活力和创造力，我们期待着您能为我们的前进贡献出您的意见和建议。同时，我们也在等待着您的加入。

我们的联系方式：

电 话：(010) 68134545      68131648

电子邮件：support@fecit.com.cn

飞思在线：<http://www.fecit.com.cn>      <http://www.fecit.net>

通用网址：计算机图书、飞思、飞思教育、飞思科技、FECIT

电子工业出版社计算机研发部



在 IT 行业人们常常说的一句话是，技术的飞速发展给我们带来的惟一问题是其本身发展太快了。我们国家正处于全球文化产业及创意产业日新月异大格局中，电子游戏、网络游戏的发展速度之快，更是令人眼花缭乱，目不暇接。文化的移植和技术的嫁接，能使得一款网络游戏的代理催生一家上市公司和它的主人雄踞富豪榜的奇迹。外来的文化和异国的精品，已将我们的游戏玩家的口味吊得居高不下，也将他们的眼光抬得近乎刁钻。他们拭目以待，期盼着一款又一款我们民族自己的游戏产品有着一席之地，渐成气候而终将席卷全球。消费市场的成熟和需求的超前已经迫切呼唤属于中华民族自己的游戏精品，其瓶颈集中体现在人才匮乏方面，于是与国际接轨的配套人才体系的建设成为至关重要、急待解决的问题——如何培养优秀的游戏设计师和开发人员？如何开发符合市场需求的游戏产品？如何成为成功的游戏开发商？这正是我们要回答的问题。

从策划到设计，从开发到上市，一款完美的游戏出品并非易事。游戏已经不再只是打发时间的消遣，它给我们提供了一个无限伸展的内心世界，一种分享我们共同经历、体验、希望和梦想的机会。

首先，我们要了解游戏到底是什么，要回答什么是好的游戏，什么是好玩的游戏。在多元文化的市场经济的年代，我们要了解我们的游戏玩家，玩家永远是上帝。游戏就在于它好玩，而游戏的好玩就在于它的互动参与和竞技动力，能让参与者感兴趣而积极互动的游戏要素有很多，情节感受、视觉享受、难易编排、控制技能等。这一切都需要策划、设计、编写、生成、测试，形成一个以游戏产品为核心的循环链和一个完整的业界。目前在链的各个环节上，除了终端的玩家嗷嗷待哺不乏其人外，其他环节就人才稀少，多处空白。其实我们真正缺乏的不是人才，而是缺少掌握了该专业知识的行家。为了填补这些空白，让那些有潜力的人们尽快获得他们所需要的知识技能，我们特编写这套《游戏学院经典书丛》。

研究探讨游戏的要素有许许多多，仅从设计考虑就有两大方面：艺术和技术。任何一套好的游戏都要依靠精密完善的策划。在开发过程中，又有许多的讲究，流程的安排，编程的方法，制作的技巧等。因此，这套丛书主要包括六个模块：基础理论、开发技术、图形技术、架构设计、剧本策划、营运管理，已形成一个完整的体系。本套丛书从游戏项目整体思想与系统设计、游戏程序开发、游戏用户界面和视听艺术等方向切入，涵盖了游戏产业领域的几大方向，覆盖了初、中、高三个层次，涉及领域广泛，纵深适宜。参加丛书编写的作者和编审委员会有来自美国游戏界的资深专家，韩国的游戏开发专家，中国台湾的游戏开发团队，多次获奖的国内外游戏美术设计师，又有来自国内外著名高校中专门从事游戏研究和教学的学者教授。

随着电脑硬件系统的改良，芯片由 32 位提升到 128 位，未来的电子和网络游戏将是如何的发展趋势？编者认为，它必将是更智能化、更艺术性、更人性味，因而更酷。在高性能的硬件设备的支持下，游戏的驱动引擎更具威力，控制功能更加完备，在一个错综复杂的极度扩张的游戏大世界中，带着超乎常人智慧的人工智能的应对力，使得所有要素包括艺术的风格、情感的互动、故事的铺展、视觉的表现、用户的界面，融合为有机的一体，给人们带来更丰富的娱乐和体验。好的游戏给人们的生活带来了健康愉快的补充，唯美的艺术享受，潜默的教育功效，时尚的情感陶冶。市场总是要开张的，如果没有足够的优质游戏，那么市场就不免会被粗劣的游戏所充斥。面对着广大成长中的青少年，谁掌握着游戏的导向权更是至关重要。

除去市场价值的意义，电子游戏包括网络游戏的发展的必要性还在于对我国文化产业发展的重要性。游戏作为一种特殊的创意文化产品形态，是文化产业链中的不可缺少的一环。创意产业是文化产业中最具创造性和先导性的核心组成部分，在创意产业已成为其他产业核心的新经济时代（美国创意产业占 GNP 的 70%，加拿大占 GDP 的 60%），现在全球创意经济的产值每天达到 220 亿美元，中国的经济转型在世界的整体发展趋势中也将从过去的中国制造逐步转向未来的中国创造。我们希望有更多的业界有识之士能关注这一新的发展领域，投入自己的聪明才智、精力能量去解决开创游戏业的一道道难题；我们也希望在这套由游戏专家群体精心编写的丛书中，你能找到大部分解决问题的方案，无论你想成为这一产业的经营者，还是设计创作人员，或是玩家。特别是对于从事游戏专业领域的教学科研和教育培训相关的教师和学生，这是一套完整的教科书。欢迎政府组织、教育机构、业界同仁和专业人士提出宝贵的建议和批评，共同参与，不断完善游戏教育体系，为我国的游戏产业的健康发展做出义不容辞的贡献。

丛书主编：肖永亮

本书的诉求理念是：以 Visual C++ 为主，辅以 DirectX9 来设计游戏，主要针对想学习游戏设计的初学者为出发点，让完全不了解 DirectX9 及游戏设计的初学者，可以通过本书深入浅出的概念与实作逐步地实现自己制作游戏的梦想。

对于许多刚踏入游戏设计领域的初学者来说，想要利用 Visual C++ 及 DirectX9 来设计游戏，必然会遇到许多实践上的困难。基于这个理由，本书在内容的编排方式上采用循序渐进的方式，从最基本的游戏设计知识，到实际范例的撰写，让初学者能轻松地学会 DirectX9 的应用与游戏设计的概念。

看了本书之后您能够清楚地了解游戏设计的基本知识。如 Visual C++ 的基本操作、游戏书画的坐标系统、如何设定游戏的主要架构等，这些知识对于刚进入游戏设计的初学者有极大的帮助。

当今的时代，游戏设计已经成为许多玩家的梦想。笔者相信有许多玩家凭着自己的崇高游戏理念与构思，想在游戏设计的领域里闯出一片天空，但自己却空无能力将这种构思理念变成游戏中的故事、剧情、玩法与机制，以至于造成半途而废的人也就越来越多了。

基于这个出发点，为了让这些人能够重新拾起游戏设计的信心，本书将以各种层面的技术与范例，来告诉您游戏设计的知识与技巧，邀您一同进入游戏设计这个奇妙的领域，把这些认为不可能的梦想实现吧！

本书编排的方式完全以学校教学的需求与初学者自学的方便性来考虑，内容的陈述平铺直叙，加上几个实例教学，所以，本书非常适合游戏开发人员及游戏相关专业师生学习使用。

此外，我们也讲解了各类设计技巧及技术，包括电脑游戏的画面显像、如何结合 DirectX 来设计游戏及游戏设计中会使用到的各类理论与实例。在游戏实作部分，我们以常见的游戏作为书中的游戏范本，包括俄罗斯方块、抢娃娃游戏，并利用简单又明了的范例程序代码，来让您可以轻松的学习游戏设计的技巧与乐趣。笔者深信这是一本非常实用的游戏设计宝典，最后，愿本书能够成为您进入游戏设计领域的敲门砖。

荣钦科技 敬上



# 目 录

第1章 Windows API 程序快速入门.....	1
1.1 VC++与 Windows API .....	1
1.2 构建游戏设计的舞台 .....	2
1.2.1 建立程序项目 .....	2
1.2.2 程序架构说明 .....	5
课后重点整理 .....	9
第2章 游戏画面绘图 .....	11
2.1 基本屏幕绘图 .....	11
2.1.1 坐标与 DC .....	11
2.1.2 画笔与画刷 .....	13
2.1.3 GDI 绘图函数 .....	17
2.1.4 绘制位图 .....	25
2.2 游戏画面特效制作 .....	30
2.2.1 透明效果 .....	30
2.2.2 半透明效果 .....	34
2.2.3 透明半透明效果 .....	41
2.3 游戏地图制作 .....	46
2.3.1 平面地图贴图 .....	46
2.3.2 斜角地图贴图 .....	50
2.3.3 景物贴图 .....	55
课后重点整理 .....	59
第3章 游戏动画技巧 .....	61
3.1 基础动画显示 .....	61
3.1.1 定时器的使用 .....	61
3.1.2 游戏循环 .....	65
3.1.3 透明动画 .....	69
3.2 动画显示问题 .....	72
3.2.1 贴图坐标修正 .....	72
3.2.2 排序贴图 .....	74
3.3 背景动画设计 .....	82
3.3.1 单一背景滚动 .....	82
3.3.2 循环背景动画 .....	85
3.3.3 多背景循环动画 .....	89
课后重点整理 .....	93
第4章 游戏输入消息处理 .....	95
4.1 键盘输入消息 .....	95
4.1.1 关于 Windows 中的键盘 .....	95
4.1.2 键盘消息处理 .....	96
4.2 鼠标输入消息 .....	103
4.3 鼠标相关函数 .....	105
4.3.1 获取窗口外鼠标消息 .....	106

4.3.2	设定鼠标光标位置 .....	106
4.3.3	显示与隐藏鼠标光标 .....	107
4.3.4	限制鼠标光标移动区域 .....	107
	课后重点整理 .....	115
<b>第 5 章</b>	<b>游戏人工智能 .....</b>	<b>117</b>
5.1	移动型游戏 AI .....	117
5.1.1	追逐移动 .....	117
5.1.2	躲避移动 .....	121
5.1.3	模式移动 .....	122
5.2	行为型游戏 AI .....	122
5.2.1	计算机角色的思考与行为 .....	122
5.2.2	搜寻迷宫出口 .....	136
5.3	策略型游戏 AI .....	147
	课后重点整理 .....	167
<b>第 6 章</b>	<b>游戏物理现象设计原理 .....</b>	<b>169</b>
6.1	物理运动 .....	169
6.1.1	匀速运动 .....	169
6.1.2	加速度运动 .....	173
6.1.3	重力 .....	174
6.1.4	摩擦力 .....	176
6.2	物体间的碰撞 .....	179
6.2.1	以范围检测碰撞 .....	179
6.2.2	以颜色检测碰撞 .....	182
6.2.3	以行进路线检测碰撞 .....	186
6.2.4	与斜面碰撞后的速度 .....	188
6.3	粒子的应用 .....	191
6.3.1	粒子的定义 .....	192
6.3.2	雪花纷飞 .....	192
6.3.3	放烟火 .....	194
	课后重点整理 .....	198
<b>第 7 章</b>	<b>进入 3D 世界 .....</b>	<b>199</b>
7.1	初探 DirectX .....	199
7.1.1	DirectX SDK 简介 .....	199
7.1.2	DirectX 的特色 .....	199
7.2	使用 Direct Graphics .....	200
7.2.1	介绍 Direct Graphics .....	200
7.2.2	如何建立 Direct Graphics 设备 .....	201
7.2.3	使用 Direct Graphics 取得绘图设备 (GDI) .....	205
7.3	使用 Direct Graphics 进行 2D 影像处理 .....	208
7.3.1	Direct Graphics 绘图引擎 .....	208
7.3.2	如何贴影像文件 .....	208
7.4	Direct Graphics 的颜色操作 .....	216
7.4.1	Direct Graphics 颜色操作流程 .....	217
7.4.2	混色操作 .....	217
7.4.3	材质基台操作 .....	220

课后重点整理.....	223
第 8 章 Direct Graphics 3D 的奇幻世界.....	225
8.1 迷人的 3D 魅力.....	225
8.1.1 三维空间概念.....	225
8.1.2 模型与顶点.....	225
8.1.3 3D 世界的环境描述.....	226
8.1.4 顶点颜色的计算方法.....	228
8.1.5 加载一个 X 文件的模型.....	228
8.2 3D 空间坐标的转换.....	232
8.2.1 Direct Graphics 坐标转换管线.....	232
8.2.2 世界环境描述.....	233
8.3.3 视角环境描述.....	235
8.3.4 投射环境描述.....	236
8.4 Direct Graphics 的色彩计算.....	238
8.4.1 颜色的决定因素.....	238
8.4.2 发射光的设定方式.....	238
8.4.3 表面材质的设定方法.....	241
课后重点整理.....	243
第 9 章 DirectSound 的使用方式.....	245
9.1 开始建立 DirectSound.....	245
9.1.1 建立 DirectSound 的第一步.....	245
9.1.2 DirectSound 对象的建立.....	246
9.1.3 设定程序协调层级.....	247
9.1.4 缓冲区的基本概念.....	247
9.1.5 建立主缓冲区.....	248
9.1.6 WAVE 声音文件的加载.....	250
9.1.7 建立次缓冲区.....	253
9.1.8 加载声音到次缓冲区.....	254
9.2 声音的播放与控制.....	255
9.2.1 播放声音功能.....	255
9.2.2 制作混音功能.....	255
9.2.3 控制声音功能.....	257
9.3 3D 音效的实际演练.....	262
9.3.1 认识 3D 音效.....	262
9.3.2 建立倾听者功能.....	264
9.3.3 建立发声者.....	265
课后重点整理.....	269
第 10 章 DirectInput 的使用方法.....	271
10.1 建立 DirectInput 程序.....	271
10.1.1 开始建立 DirectInput 程序.....	271
10.1.2 建立 DirectInput 对象.....	272
10.1.3 建立输入装置对象.....	273
10.1.4 资料格式的设定.....	273
10.1.5 设定程序协调层级.....	274
10.1.6 输入装置的调用方法.....	274



10.2 键盘与鼠标输入的取得方法 .....	275
10.2.1 键盘输入的取得 .....	275
10.2.2 取得鼠标输入 .....	278
10.3 使用摇杆功能 .....	282
10.3.1 取得摇杆装置 .....	282
10.3.2 摇杆组件的列举方法 .....	284
10.3.3 摇杆输入的取得 .....	288
10.3.4 设定无效范围 .....	290
课后重点整理 .....	292
<b>第11章 威力强大的 DirectPlay 和 DirectShow .....</b>	<b>293</b>
11.1 DirectPlay 初体验 .....	293
11.1.1 DirectPlay 的使用时机 .....	293
11.1.2 DirectPlay 的网络拓扑 .....	293
11.1.3 网络联机游戏的构成 .....	295
11.1.4 DirectPlay 的组成模式 .....	296
11.1.5 联机程序范例介绍 .....	300
11.2 DirectShow 的多媒体功能 .....	310
11.2.1 DirectShow 的架构 .....	310
11.2.2 播放影片功能 .....	312
11.2.3 播放 MP3 .....	313
课后重点整理 .....	313
<b>第12章 小游戏设计实例 .....</b>	<b>315</b>
12.1 俄罗斯方块游戏轻松做 .....	315
12.2 抢娃娃游戏 .....	330
<b>附录 A DirectDraw 制作游戏秘籍大公开 .....</b>	<b>349</b>
A.1 程序中的各个自定义函数 .....	349
A.1.1 初始化与建立 DirectX 对象 .....	349
A.1.2 建立 DirectDraw 幕后暂存区 .....	353
A.1.3 建立 DirectSound 次缓冲区 .....	355
A.1.4 设定颜色键函数 .....	357
A.2 绚丽的电流急急棒 .....	357
A.2.1 游戏功能介绍 .....	358
A.2.2 游戏功能设计方法 .....	359
A.2.3 程序内容说明 .....	360
A.3 太空射击游戏 .....	366
A.3.1 游戏功能介绍 .....	366
A.3.2 滚动背景的设计 .....	368
A.3.3 怪物的产生与移动 .....	368
A.3.4 子弹的产生 .....	369
A.3.5 检测碰撞的方法 .....	369
A.3.6 程序编写的方法 .....	372
<b>附录 B 专业词汇 .....</b>	<b>387</b>
<b>附录 C 常用 Windows 虚拟键表 .....</b>	<b>395</b>

# 第 1 章 Windows API 程序快速入门

## 1.1 VC++与 Windows API

由于计算机游戏程序结合了大量声音、影像数据的运算处理，因此，流畅度是程序运行时相当重要的一个基本要求。为了达到这项要求，目前一般的大型商业游戏软件开发模式，大多采用 Visual C++（简称 VC++）程序开发工具与 Windows API（Application Program Interface）程序架构来编写，以提高游戏程序运行时的效率。

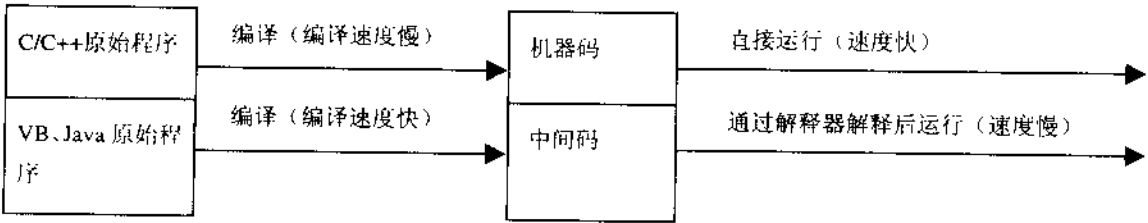
在这一节的内容里，我们首先要来谈谈 VC++ 与 Windows API 在开发游戏程序上所具备的优点，在稍后的内容中会更详细地介绍 Windows API 程序的基本构架及处理外界消息的方式，以最简单易懂的方式引导您快速登上游戏程序设计的舞台。

C++ 是拥有优良传统的程序语言，而 VC++ 则是微软公司开发出一套适用于 C/C++ 语法的程序开发工具。在 VC++ 开发环境中，编写 Windows 操作系统平台的窗口程序有两种不同的程序架构：一种是微软在 VC++ 中所加入的 MFC（Microsoft Foundation Class library）架构，MFC 是一个庞大的类型函数库，其中提供了完整开发窗口程序所需的对象类型与函数，常用于设计一般的应用软件程序；另一种是本书所介绍的 Windows API 架构，使用 Windows API 来开发上述的应用软件程序并不容易，但用在设计游戏程序上却相当简单且具有较优越的运行性能。

因此，不论开发一般应用软件还是游戏程序，VC++ 在程序开发领域中的使用率均相当的高，下面我们就来说明 VC++ 在游戏程序开发上所具备的优点。

### 1. 优越的速度表现

C++ 程序编译后的文件是可直接运行的机器码，而其他程序语言（如 VB 和 Java 这类程序）编译后产生的是一种所谓的“中间码”。运行中间码时，系统内必须存在解释该程序语言的“解释器（Interpreter）”，用做同步翻译工作。解释器解释中间码时，本身会加载到内存中占用部分内存，且同步翻译中间码的过程也会浪费时间，因此，运行这类程序时要比可直接运行的机器码缓慢得多，如图 1-1 所示，这就是 VC++ 程序有较优越运行速度的主要原因之一。



C/C++、VB、Java 程序编译和运行流程比较

图 1-1

## 2. 弹性管理资源与内存

在 VC++ 的开发环境中，程序资源及内存管理方面都具有相当的弹性。

在资源管理部分，通常是通过一个句柄来使用该项资源。这里所指的资源，可能是窗口、设备、图像和声音等对象。

在内存管理部分，C/C++ 语言本身就具备内存管理的功能，除了可通过指针进行内存的存取和配置之外，还提供了完整的内存管理相关函数。

由于游戏程序使用了大量的多媒体数据，运行时会占用不少内存，因此，若程序设计师能够弹性有效地来管理资源和内存，将可大大降低硬件要求并提高游戏程序本身的性能。

## 3. 易于使用 Windows API

Windows API 是 Windows 操作系统提供的动态链接函数库（通常以“.DLL”的文件格式存在于 Windows 系统中），Windows API 中包含了 Windows 的内核及所有应用程序所需要的功能。

Windows 操作系统发展至今，Windows API 主要可分为 Win16（Windows 3.1 以前）以及 Win32（Windows 95 以后）两种版本，不同版本 Windows 系统间 API 的内容或多或少有些差异，但都以向下兼容为原则，例如，在 Windows 98 下使用 Windows 98 API 所开发的窗口程序，在 Windows 2000 或 Windows XP 系统上仍然可正常运行。

如果您曾使用 VB 写过窗口程序的话，那么您可能会清楚，一般在 VB 程序中，若要调用 Windows API 的函数，必须先完成声明的操作。但在 VC++ 开发环境下，不论采用 MFC 还是 Windows API 的程序架构，只要我们在项目中设定好所要链接的函数库并引用正确的头文件，那么在程序中使用 Windows API 的函数就跟使用 C/C++ 标准函数库一样容易。

通过以上说明，相信您对于为何使用 VC++ 来开发游戏程序有了一定的认识。以具有面向对象特性且运行性能佳的 C++ 程序为主体，配合 Windows 操作系统本身的 API，以及其他如 DirectX 与 OpenGL 技术来发展游戏程序。相信在未来的一段时间内，VC++ 依然会是 Windows 平台上游戏开发的最佳利器！

### VC++ 技巧 动态链接

动态链接（Dynamic Linking）是指在程序运行阶段，真正调用外部函数时才进行链接（注：将程序代码中调用函数的指针指向外部函数所在的地址）的操作。

## 1.2 构建游戏设计的舞台

本节中开始试着使用 VC++ 来建立一个 Windows API 架构的基本程序项目，程序运行时会产生一个简单的程序窗口，而本书中的所有范例都将以此个项目为基础来讲解游戏的功能。

### 1.2.1 建立程序项目

如果您是第一次接触 VC++，或许还不清楚建立 VC++ 程序项目的方式，不过没关系，在这里先以 VC++ 项目向导做个建立简单项目的示例，可以按照下面的操作步骤来完成这项工作，如图 1-2~图 1-4 所示。

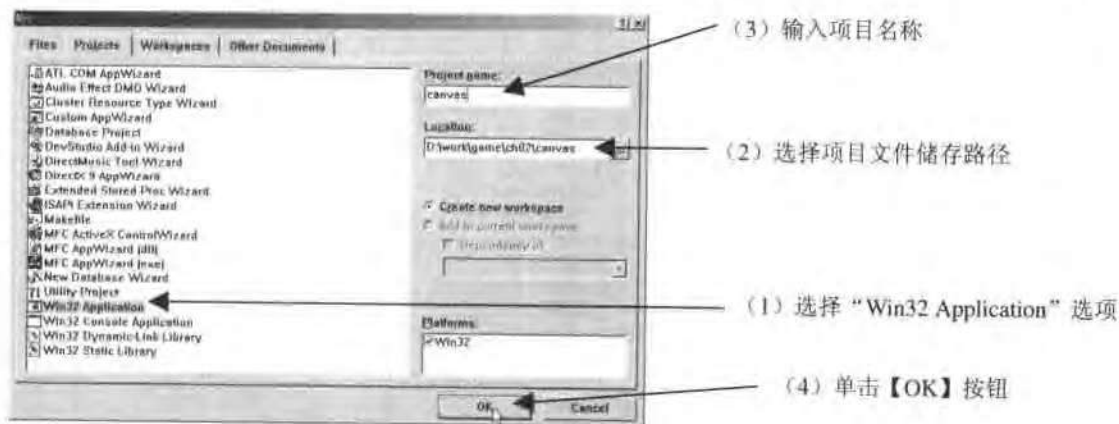


图 1-2

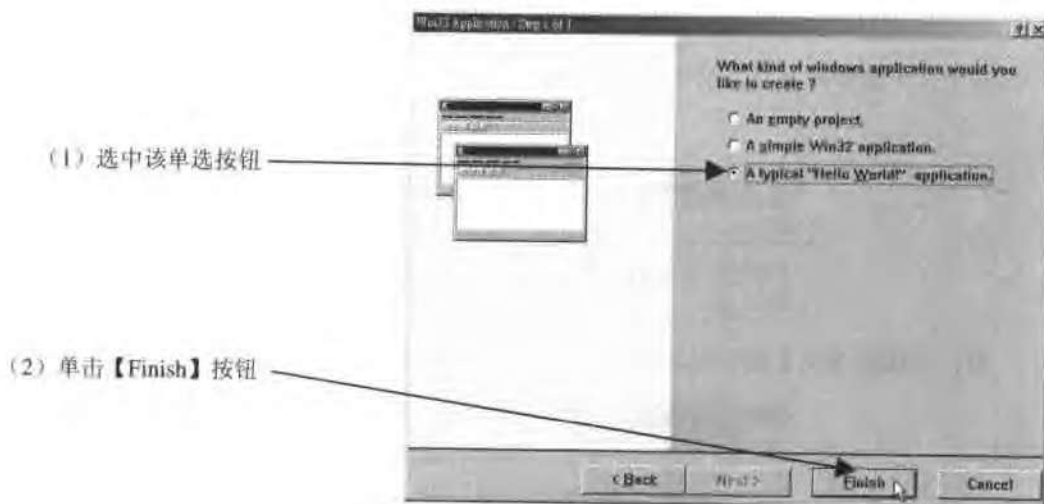


图 1-3

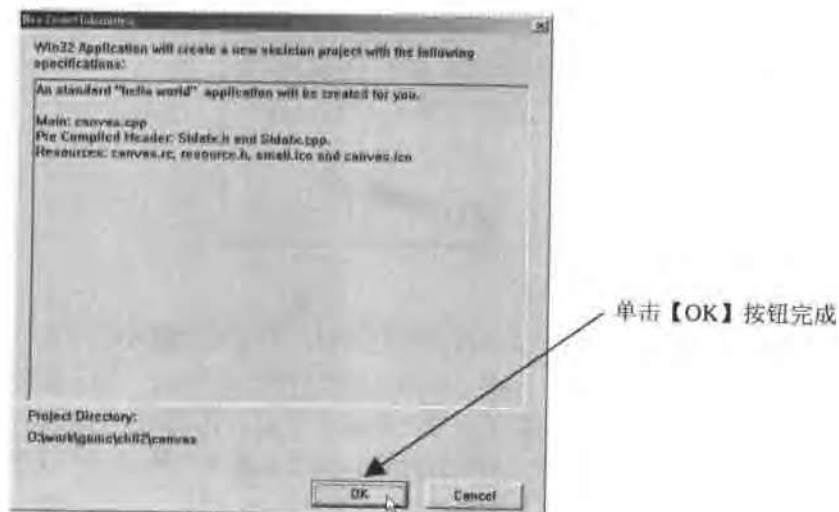


图 1-4

在项目建立完成后，查看工作区中的文件列表，可看到如图 1-5 所示的内容。



图 1-5

其中几个重要的文件说明如表 1-1 所示。

表 1-1

文件名称	说 明
canvasp.cpp	主程序文件，其中包含整个项目的主程序 WinMain
canvas.rc	资源文件，定义了整个项目所使用的资源
StdAfx.h	标头文件，其中引用整个项目所需的头文件

可以直接按【F5】键运行这个项目，运行画面如图 1-6 所示。



图 1-6

可以看到，这是一个简单的应用程序，只包含菜单和说明对话框。不过通常游戏程序并不会使用一般应用程序所需要的菜单和按钮等窗口组件。因此，可以将这个程序加以修改，使其能符合实际设计游戏的需求。这里将程序代码进行简化，删除不需要的资源，并加上中文批注，修改后的程序可作为后面介绍游戏设计时的基本骨架项目，修改后的程序运行画面如图 1-7 所示。可以在书中所附光盘第 1 章的数据文件夹中找到“canvas”这个基本项目。

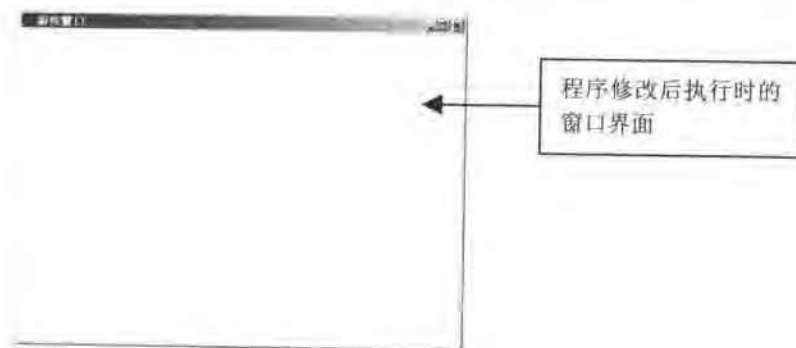


图 1-7

有了设计游戏的框架程序，接下来要谈谈这个程序的基本架构，您将会慢慢熟悉 Windows API 窗口程序的写法，并了解如何进一步来扩展我们的游戏程序。

## 1.2.2 程序架构说明

上一小节中建立的项目便是一个标准 Windows API 的程序架构，主程序文件“canvas.cpp”由表 1-2 中几个重要的函数所构成。

表 1-2

函数名称	说 明
WinMain	主程序，程序起始点
WndProc	自定义函数，处理程序消息
MyRegisterClass	自定义函数，注册窗口类别
InitInstance	自定义函数，建立及显示窗口的初始化函数

下面来解说这些函数的内容与功能。

### 1. WinMain 函数

主程序，也就是整个项目程序开始运行的地方，如同 Console 程序中的 main() 函数。

```
1  //****主程序*****
2  int APIENTRY WinMain(HINSTANCE hInstance,
3                      HINSTANCE hPrevInstance,
4                      LPSTR lpCmdLine,
5                      int nCmdShow)
6  {
7      MSG msg;
8
9      MyRegisterClass(hInstance);
10
11     //运行初始化函数
12     if (!InitInstance (hInstance, nCmdShow))
13     {
14         return FALSE;
```



```

15 }
16
17 //消息循环
18 while (GetMessage(&msg, NULL, 0, 0))
19 {
20     TranslateMessage(&msg);
21     DispatchMessage(&msg);
22 }
23
24 return msg.wParam;
25 }

```

## 程序说明

(1) 第 9 行: 调用 MyRegisterClass() 函数, 向系统注册窗口类别, 输入参数 “hInstance” 是目前程序运行个体的对象代码。

(2) 第 12~15 行: 调用 InitInstance() 函数, 进行初始化操作。

(3) 第 18~22 行: 程序通过此消息循环来获取消息, 并进行必要的键盘消息转换, 而后将控制权交给操作系统, 由操作系统决定该由哪个程序的消息处理函数处理消息。这个循环使用表 1-3 中的 3 个 API 函数。

表 1-3

函数名称	说 明
GetMessage (第 18 行)	获取程序消息
TranslateMessage (第 20 行)	转换伪码及字符
DispatchMessage (第 21 行)	将控制权交给系统, 再由系统决定负责处理消息的程序

## 2. MyRegisterClass 函数

在建立程序窗口的实体之前, 必须先定义一个窗口类别, 其中包含所要建立窗口的相关信息, 并向系统注册。这里的 MyRegisterClass() 函数就是进行定义及注册窗口类别的函数。

```

1  //****定义及注册窗口类别函数*****
2  ATOM MyRegisterClass (HINSTANCE hInstance)
3  {
4      WNDCLASSEX wcex;
5
6      wcex.cbSize = sizeof(WNDCLASSEX);
7      wcex.style      = CS_HREDRAW | CS_VREDRAW;
8      wcex.lpfnWndProc = (WNDPROC)WndProc;    //消息处理函数
9      wcex.cbClsExtra  = 0;
10     wcex.cbWndExtra  = 0;
11     wcex.hInstance   = hInstance;
12     wcex.hIcon        = NULL;
13     wcex.hCursor      = NULL;
14     wcex.hCursor      = LoadCursor(NULL, IDC_ARROW);
15     wcex.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
16     wcex.lpszMenuName  = NULL;
17     wcex.lpszClassName = "canvas";          //类别名称

```

```
18 wcex.hIconSm      = NULL;
19
20 return RegisterClassEx(&wcex);
21 }
```

#### 程序说明

- (1) 第4行：声明一个窗口类别“WNDCLASSEX”和结构“wcex”。
- (2) 第6~18行：定义“wcex”结构的各项信息，其中设定消息处理函数（lpfnWndProc）为“WndProc”，类别名称（lpszClassName）为“canvas”。
- (3) 第20行：调用 RegisterClassEx()函数注册类别，返回一个“ATOM”形态的字符串，此字符串即为类别名称“canvas”。

### 3. InitInstance 函数

在这一初始函数中，按照前面所定义的窗口类别来建立并显示实际的程序窗口。

```
1  //****初始化函数*****
2  // 1.存储 instance handle 于全局变量中
3  // 2.建立并显示主窗口
4  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
5  {
6      HWND hWnd;
7
8      hInst = hInstance;
9
10     hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
11                        0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
12
13     if (!hWnd)
14     {
15         return FALSE;
16     }
17     MoveWindow(hWnd, 10, 10, 600, 450, true);
18     ShowWindow(hWnd, nCmdShow);
19     UpdateWindow(hWnd);
20
21     return TRUE;
22 }
```

#### 程序说明

- (1) 第10行：调用 CreateWindow()函数来建立一个窗口对象，所输入的第1个参数就是窗口建立所依据的类别名称，也就是前面程序所定义的“canvas”。
- (2) 第17~19行：设定窗口的显示位置及窗口大小，然后绘制在显示设备上，这里使用表1-4中的3个API函数。

表 1-4

函数名称	说 明
MoveWindows (第 18 行)	设定窗口显示的位置及窗口大小
ShowWindow (第 19 行)	设定窗口显示时的状态
UpdateWindow (第 20 行)	将窗口绘制于显示设备上

## 4. WndPro 函数

在前面定义类别的时候把 WndPro 定义为消息处理函数，也就是当某些外部消息发生时，会按照消息的类型来决定该如何进行处理。

此外，WndPro 函数也是一个所谓的“回叫函数 (CALLBACK)”，简单地说，回叫函数是属于 Windows 操作系统所调用的函数，而非程序本身所调用的函数，这是因为 Windows 是一个多任务的作业环境，在同一时刻可能会有多个程序正在运行，而当某一事件发生时，有可能每一个程序都会接收到此消息，因此系统必须去判断该由哪个程序来进行处理，然后再调用该程序的消息处理函数。

```

1  //****消息处理函数*****
2  LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
    lParam)
3  {
4      PAINTSTRUCT ps;
5      HDC hdc;
6
7      switch (message)
8      {
9          case WM_PAINT:                //窗口重绘消息
10             hdc = BeginPaint(hWnd, &ps);
11             EndPaint(hWnd, &ps);
12             break;
13          case WM_DESTROY:              //窗口结束消息
14             PostQuitMessage(0);
15             break;
16          default:                      //其他消息
17             return DefWindowProc(hWnd, message, wParam, lParam);
18      }
19      return 0;
20  }
```

### 程序说明

(1) 第 7 行：判断消息类型。

(2) 第 9~12 行：处理窗口重绘消息 (WM\_PAINT)，处理步骤将在后面介绍屏幕绘图方式时再做说明。

(3) 第 13~15 行：处理窗口结束消息 (WM\_DESTROY)，调用 PostQuitMessage() 函数发送窗口结束消息给系统，通知系统结束目前程序的运行，然后由系统处理程序结束的后续操作。

(4) 第 16~17 行：DefWindowProc() 函数调用预设的系统函数来处理程序本身不处理的消息。

本章讨论了 VC++ 在开发游戏程序上所具备的优点，并快速地浏览了 Windows API 程序的基本

构架。下一章将从屏幕绘图开始，讲解更多更有趣的设计游戏程序的技巧。

## 课后重点整理

- 在 VC++ 的开发环境中，编写 Windows 操作系统平台的窗口程序有两种不同的程序架构：一种是微软在 VC++ 中所加入的 MFC (Microsoft Foundation Class library) 架构，另一种是 Windows API 架构。
- VC++ 在游戏程序开发上具备优越的速度表现、弹性管理资源与内存、易于使用 Windows API 等优点。
- C++ 程序编译后的文件是可直接运行的机器码。
- 在 VC++ 的开发环境中，在资源管理部分，通常是通过一个句柄来使用该项资源的。
- 在内存管理部分，C/C++ 语言本身就具备内存管理的功能，除了可通过指针进行内存的存取、配置之外，还提供完整的内存管理相关函数。
- Windows API (Application Program Interface) 是 Windows 操作系统所提供的动态链接函数库 (通常以“.DLL" 的文件格式存在于 Windows 系统中)，Windows API 中包含了 Windows 的内核及所有应用程序所需要的功能。
- Windows 操作系统发展至今，Windows API 主要可分为 Win16 (Windows 3.1 以前) 及 Win32 (Windows 95 以后) 两种版本，不同版本 Windows 间 API 的内容或许有些差异，但都是以向下兼容为原则的。
- 动态链接 (Dynamic Linking) 是指在程序运行阶段，真正调用到外部函数时才进行链接操作。
- VC++ 通过提供项目向导，可方便使用者建立程序项目。

### 课后练习

1. 项目建立完成后，查看工作区中可以看到一些文件列表，请解释表 1-5 中重要文件的功能。

表 1-5

文件名称	说 明
canvas.cpp	
canvas.rc	
StdAfx.h	

2. 请练习利用项目向导，体会自行建立一个新项目的过程。
3. 本书中主程序文件“canvas.cpp”由表 1-6 中的几个重要的函数所构成，请简单说明这些函数的主要功能。

表 1-6

函数名称	说 明
WinMain	
WndProc	
MyRegisterClass	
InitInstance	



## 第 2 章 游戏画面绘图

### 2.1 基本屏幕绘图

GDI (Graphics Device Interface) 中文可译为“图形设备接口”，是 Windows API 中相当重要的一个成员，它掌管了所有显像设备的图像显示及输出功能，缺少了它，Windows 系统将不会像现在所使用的图形操作环境，而游戏程序既然运用了大量的影像图形处理，自然也就少不了 GDI。

这一节将讲解 GDI 函数的使用及窗口粘贴图像的技巧，渐进式地学习如何来绘制所需要的游戏画面。

#### 2.1.1 坐标与 DC

在实际利用 GDI 来进行程序绘图之前，这一小节里先来了解一下几个屏幕绘图的基本概念。

##### 1. 屏幕区、窗口区与内部窗口区

对一个游戏程序来说，不论采用全屏幕的模式还是单纯的窗口模式，都必须建立一个窗口。当窗口建立之后，显示的屏幕上便划分为 3 个区域，即屏幕区 (Screen)、窗口区 (Window) 与内部窗口区 (Client)，如图 2-1 所示。

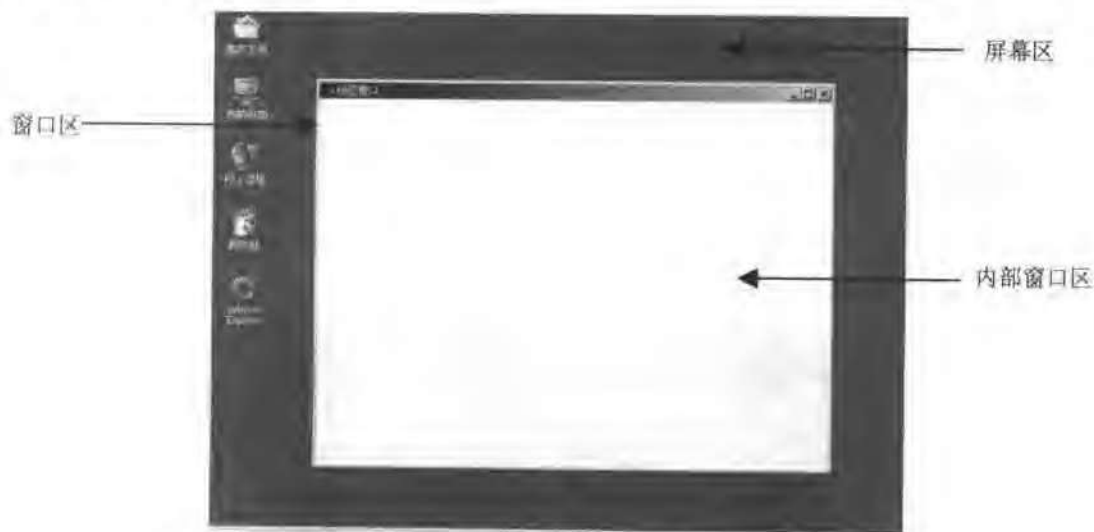


图 2-1

屏幕区的区域大小视使用者计算机所设定的显示分辨率而定，一般在程序中都是以像素 (pixel) 当做坐标及长度的单位，例如若使用者的计算机目前的显示分辨率为  $800 \times 600$  像素，则屏幕区矩形的大小即为  $800 \times 600$  像素。



### 2. Device Context

Device Context (设备内容) 一般简称为 DC, 就绘图的观点来说, DC 就是程序可以进行绘图的地方。举例来说, 如果要在整个屏幕区上绘图, 那么 Device (设备) 就是屏幕, 而 DC 就是屏幕区上的绘图层。相同的道理, 如果要在窗口中绘图, 那么 Device 就是窗口, DC 就是窗口上可以绘图的地方, 也就是内部窗口区。

回顾前面消息处理函数中关于处理窗口重绘消息的程序片段。

```
1 case WM_PAINT: //窗口重绘消息
2   hdc = BeginPaint(hWnd, &ps);
3   EndPaint(hWnd, &ps);
4   break;
```

其中, 调用 `BeginPaint()` 函数开始进行窗口重绘的动作, 而调用 `EndPaint()` 函数则是结束所有绘图动作。程序代码中调用 `BeginPaint()` 时会返回一个 DC 对象 `hdc`, 此对象所代表的是窗口 (`hWnd`) DC, 也就是内部窗口区。当窗口重绘消息发生时, 可以在第 2 行和第 3 行程序代码之间加入要在内部窗口上进行绘图的动作。

此外, 在处理 `WM_PAINT` 消息之外的地方, 若要取得窗口的 DC, 必须调用下面的这个函数。

```
HDC GetDC(HWND hWnd); //取得DC
```

其中, 所输入的参数是窗口的 “handle”, 而 handle 又是什么? handle 是 Windows 系统中用来识别各种不同资源的一个句柄, 而且每一项资源的 handle 都是惟一的。根据 handle, Windows 可以快速且正确地找到所要使用的资源。

这里要提醒注意的一点是, 若使用 `GetDC()` 函数取得窗口 DC 后, 当不使用时必须将它释放, 否则其他应用程序将无法使用。对应于 `GetDC()` 的释放 DC 的函数为 `ReleaseDC()`。

```
int ReleaseDC(HWND hWnd, HDC 释放DC名称); //释放DC
```

若上面这个函数运行成功, 则会返回整数 “1”, 若失败则返回 “0”。

### 3. 坐标系统

在了解 DC 的概念以后, 接着再来说明与实际图形绘制位置相关的坐标系统。在此以屏幕区为例, 绘图时其二维平面的坐标系统如图 2-2 所示。

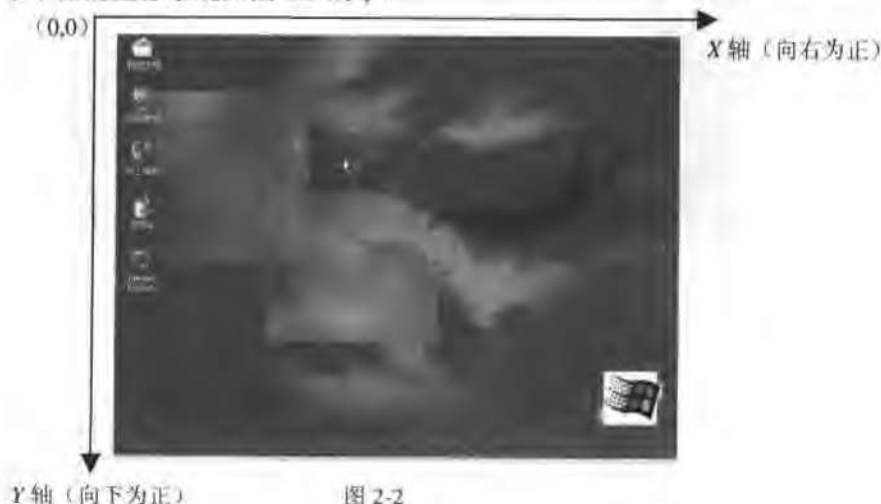


图 2-2

以左上角的坐标点为原点，屏幕上的任何一个位置都可以某一点坐标来表示，称为屏幕坐标。若绘图的区域是内部窗口区，那么坐标原点所在的位置就不同了，变成了在内部窗口最左上角的点，如图 2-3 所示。



图 2-3

#### 4. 坐标转换

从屏幕区的角度或者从内部窗口的角度来描述一个点的坐标是不一样的，例如，内部窗口坐标的原点 (0,0)，其屏幕坐标可能会是 (10,50)。对于坐标的转换，GDI 中提供了相关的函数，这将在稍后的应用中介绍。

### 2.1.2 画笔与画刷

画笔与画刷都是 GDI 中所定义的图形对象，画笔是线条的样式，画刷则是封闭图形内部填充的样式。可以自定义绘图所用画笔及画刷的样式，系统预设画笔的样式为 BLACK\_PEN，画刷的样式为 NULL\_BRUSH。图 2-4 和图 2-5 是使用默认值及自定义画笔和画刷所绘制的填充矩形。

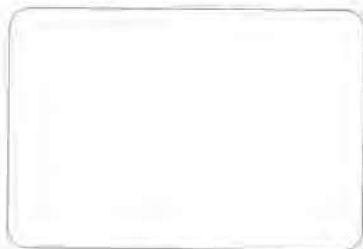


图 2-4



图 2-5

要自定义画笔或画刷，可以使用下列 3 个 API 函数。

HPEN CreatePen(int 样式,int 宽度,COLORREF 颜色);	//建立画笔
HBRUSH CreateHatchBrush(int 样式,COLORREF 颜色);	//建立阴影画刷
HBRUSH CreateSolidBrush(COLORREF 颜色);	//建立单色画刷

从上面的 3 个 API 函数可以看出，其返回值都是以英文字母“H”开头，“H”在这里代表的就是前面所提过的“handle”。

建立新画笔与画刷之后，必须在所要进行绘图的 DC 中选用它们，才会产生预期的画笔及画刷效果，选用的函数如下。

```
HGDIOBJ SelectObject(HDC hdc, HGDIOBJ GDI 对象);    //选用 GDI 对象
```

上面的这个 SelectObject()函数很重要，后面的内容里会经常使用到它，它所输入的第 2 个参数就是 GDI 对象的 handle，而返回值则是前一次所使用的 GDI 对象。除了这一小节所介绍的画笔、画刷是属于 GDI 对象外，其他的 GDI 对象还有：位图、字体、区域及调色板。

GDI 对象一经建立便会占用部分内存，一旦不使用的時候，务必将它们删除，删除函数如下。

```
BOOL DeleteObject( HGDIOBJ GDI 对象);    //删除 GDI 对象
```

若删除对象成功，则会返回布尔值“TRUE”，若失败则返回“FALSE”。

上面以画笔及画刷说明了 GDI 对象使用的基本过程：建立→选用→删除。下面先来看一个画笔与画刷的范例，相信您应该会慢慢熟悉 GDI 对象的应用。

## 》》范例 ch2\_1

建立 7 种系统所提供的画笔及画刷样式，将结果绘制于窗口中。

### 程序代码：全局变量声明

```
1  //全局变量声明
2  HINSTANCE hInst;
3  HPEN hPen[7];
4  HBRUSH hBru[7];
5  int sPen[7] = {PS_SOLID, PS_DASH, PS_DOT, PS_DASHDOT, PS_DASHDOTDOT, PS_NULL,
    PS_INSIDEFRAME};
6  int sBru[6] = {HS_VERTICAL, HS_HORIZONTAL, HS_CROSS, HS_DIAGCROSS, HS_FDIAGONAL,
    HS_BDIAGONAL};
```

### 程序说明

- (1) 第 3~4 行：画笔及画刷对象数组声明。
- (2) 第 5 行：指定 7 种系统所提供的画笔数组。
- (3) 第 6 行：指定 6 种系统所提供的阴影画刷数组。

### 程序代码：InitInstance()

```
1  //****初始化函数*****
2  // 建立 7 种不同的画笔及画刷对象
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {
5      HWND hWnd;
6      HDC hdc;
7      int i;
8
9      hInst = hInstance;
10
11     hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
```

```

0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
12
13 if (!hWnd)
14 {
15     return FALSE;
16 }
17
18 MoveWindow(hWnd, 10, 10, 650, 350, true);
19 ShowWindow(hWnd, nCmdShow);
20 UpdateWindow(hWnd);
21
22 for(i=0; i<=6; i++)
23 {
24     hPen[i] = CreatePen(sPen[i], 1, RGB(255, 0, 0));
25     if(i==6)
26         hBru[i] = CreateSolidBrush(RGB(0, 255, 0));
27     else
28         hBru[i] = CreateHatchBrush(sBru[i], RGB(0, 255, 0));
29 }
30
31 hdc = GetDC(hWnd);
32 MyPaint(hdc);
33 ReleaseDC(hWnd);
34
35 return TRUE;
36 }

```

#### 程序说明

在初始化函数中加入建立画笔及画刷的程序代码，并在建立窗口后，第一次调用 MyPaint() 自定义函数，将各类画笔与笔刷绘制于窗口中。

(1) 第 22~29 行：循环建立各种画笔与画刷。

(2) 第 24 行：按照 sPen[] 数组中的值建立各式画笔，CreatePen() 函数中输入的第 3 个参数 RGB(255, 0, 0) 为颜色结构，其值表示红色。

(3) 第 25~28 行：按照 sBru[] 数组中的值建立各式阴影画刷，当建立最后一种画刷 (i=6 时)，则调用 CreateSolidBrush() 函数建立一单色画刷，RGB(0, 255, 0) 值表示绿色。

(4) 第 31 行：调用 GetDC() 函数取得窗口的 DC。

(5) 第 32 行：调用 MyPaint() 自定义函数于首次窗口显示时进行图形绘制的动作，输入的参数 “hdc” 为窗口的 DC。

(6) 第 33 行：调用 ReleaseDC() 函数，释放所占用的 DC。

#### 程序代码：MyPaint()

```

1  //****自定义绘图函数*****
2  // 以各式画笔及画刷绘制线条与填充矩形
3  void MyPaint(HDC hdc)
4  {
5      int i, x1, x2, y;
6

```

```

7 //以 7 种不同画笔绘制线条
8 for(i=0;i<=6;i++)
9 {
10     y = (i+1) * 30;
11
12     SelectObject(hdc,hPen[i]); //选用画笔
13     MoveToEx(hdc,30,y,NULL); //移到线条起点
14     LineTo(hdc,100,y); //画线
15 }
16
17 x1 = 120;
18 x2 = 180;
19
20 //以 7 种不同画刷填充矩形
21 for(i=0;i<=6;i++)
22 {
23     SelectObject(hdc,hBru[i]); //选用画刷
24     Rectangle(hdc,x1,30,x2,y); //画封闭矩形
25     x1 += 70;
26     x2 += 70;
27 }
28 }

```

## 程序说明

在这个程序里，不论窗口第一次出现，还是窗口必须重绘时（窗口移动、窗口最大最小化）都会调用这个函数来进行内部窗口绘图的动作。

(1) 第 5 行：变量声明，x1、x2、y 是函数中所用的坐标变量。

(2) 第 8 ~15 行：以各种不同画笔绘制线条，其中用到 GDI 中的两个画线函数：

```

BOOL MoveToEx(HDC hdc,int X坐标,int Y坐标,LPPPOINT 目前坐标); //移动画笔至线条起点
BOOL LineTo(HDC hdc,int X坐标,int Y坐标); //绘制线条到指定坐标

```

(3) 第 21 ~27 行：绘制封闭矩形，并选用各种不同画笔来填充矩形内部，绘制矩形的函数为：

```

BOOL Rectangle( HDC hdc, //绘制直角矩形
                int 矩形左上点X坐标,
                int 矩形左上点Y坐标,
                int 矩形右下点X坐标,
                int 矩形右下点Y坐标 );

```

## 程序代码：WndProc()

```

1 //****消息处理函数*****
2 // 1.窗口重绘消息发生时调用 MyPaint()
3 // 2.窗口结束消息发生时删除 GDI 对象
4 LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
    lParam)
5 {
6     PAINTSTRUCT ps;
7     HDC hdc;
8     int i;

```

```
9
10 switch (message)
11 {
12     case WM_PAINT:                //窗口重绘消息
13         hdc = BeginPaint(hWnd, &ps);
14         MyPaint(hdc);
15         EndPaint(hWnd, &ps);
16         break;
17     case WM_DESTROY:              //窗口结束消息
18         for(i=0;i<=6;i++)
19         {
20             DeleteObject(hPen[i]); //删除画笔
21             DeleteObject(hBru[i]); //删除画刷
22         }
23         PostQuitMessage(0);
24         break;
25     default:                       //其他消息
26         return DefWindowProc(hWnd, message, wParam, lParam);
27 }
28 return 0;
29 }
```

#### 程序说明

(1) 第 14 行：于窗口重绘消息发生时，调用 MyPaint() 函数重绘内部窗口内容。

(2) 第 18~22 行：于窗口结束消息发生时，删除所有画笔及画刷。

#### 运行结果

程序运行结果如图 2-6 所示。

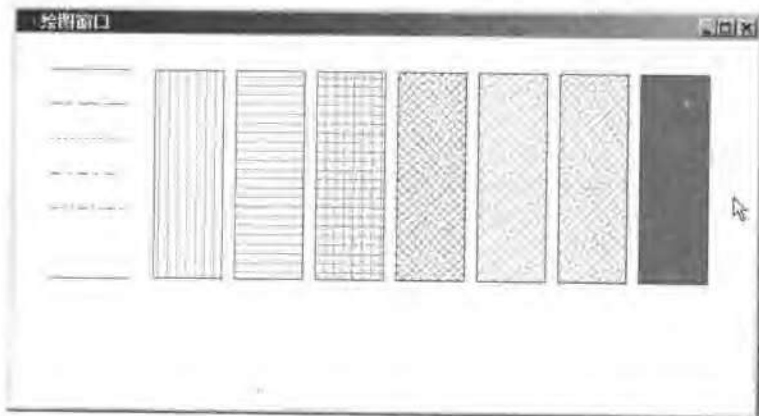


图 2-6

### 2.1.3 GDI 绘图函数

在上一小节里，以画笔及画刷作为学习屏幕绘图及 GDI 的第一堂课，并且接触了几个简单的 GDI 绘图函数。在这一小节中将介绍一些 GDI 的绘图函数，以加深您对于程序绘图的认识。



## 1. 文字输出

窗口画面上的文字输出比后面所要介绍的几个绘图函数还要重要，因为在程序设计时，可能有时需要在画面上显示一些数据和信息，以利于程序本身的追踪与排错。

下面就来看一个在窗口中输出文字的范例。

### » 范例 ch2\_2

根据鼠标的移动，将光标所在位置的坐标值显示在窗口中。

程序代码：WndProc()

```

1  //****消息处理函数*****
2  LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
3  {
4      PAINTSTRUCT ps;
5      HDC hdc;
6
7      switch (message)
8      {
9          case WM_PAINT:                //窗口重绘消息
10             hdc = BeginPaint(hWnd, &ps);
11             EndPaint(hWnd, &ps);
12             break;
13          case WM_MOUSEMOVE:            //鼠标移动消息
14             hdc = GetDC(hWnd);
15             MyPaint(hdc, lParam);
16             ReleaseDC(hWnd, hdc);
17             break;
18          case WM_DESTROY:              //窗口结束消息
19             PostQuitMessage(0);
20             break;
21          default:                      //其他消息
22             return DefWindowProc(hWnd, message, wParam, lParam);
23      }
24      return 0;
25  }

```

程序说明

(1) 第 13 行：在消息循环中加入鼠标移动消息 WM\_MOUSEMOVE 的处理。

(2) 第 14 行：调用 GetDC() 函数取得窗口 DC，与第 10 行程序调用 BeginPaint() 函数返回窗口 DC 不同的地方在于，BeginPaint() 仅用在处理窗口重绘消息上，若在其他地方要获得窗口的 DC，则使用 GetDC() 函数。

(3) 第 15 行：调用 MyPaint() 函数在窗口中输出文字，其中输入的第 2 个参数“lParam”中包含了鼠标状态的相关信息。

程序代码：MyPaint()

```

1  //****自定义绘图函数*****

```

```

2 // 显示鼠标坐标
3 void MyPaint(HDC hdc,LPARAM lParam)
4 {
5     int x,y;
6     char str[20] = "";
7
8     x = LOWORD(lParam);    //取得鼠标 X 坐标值
9     y = HIWORD(lParam);    //取得鼠标 Y 坐标值
10
11     SetTextColor(hdc,RGB(255,0,0));
12
13     TextOut(hdc,10,10,"鼠标坐标",strlen("鼠标坐标"));
14     sprintf(str,"X 坐标: %d  ",x);
15     TextOut(hdc,30,30,str,strlen(str));
16     sprintf(str,"Y 坐标: %d  ",y);
17     TextOut(hdc,30,50,str,strlen(str));
18 }

```

### 程序说明

只要光标在窗口中移动，消息处理函数就会调用 MyPaint()函数来显示光标所在位置的坐标值。

(1) 第 6 行：声明一个空字符串，用来存储要输出到屏幕上的文字。

(2) 第 8 行：调用 LOWORD()函数取得“lParam”低位字节的信息，即光标目前所在位置的 X 坐标值。

(3) 第 9 行：调用 HIWORD()函数取得“lParam”高位字节的信息，即光标目前所在位置的 Y 坐标值。

(4) 第 11 行：调用 SetTextColor()函数设定 DC 上文字输出的颜色为红色。

(5) 第 12~16 行：格式化所要显示的字符串，并调用 TextOut()函数将其输出在窗口上，TextOut()函数的使用方式如下。

```

BOOL TextOut(    HDC hdc,                //输出文字
               int 输出字符串的x坐标,
               int 输出字符串的y坐标,
               LPCTSTR 字符串指针,
               int 字符串长度);

```

### 运行结果

程序运行的结果如图 2-7 所示。



图 2-7

## 2. 多边形函数

表 2-1 中给出了 GDI 函数中关于多边形的绘图函数。

表 2-1

函数名称	说明
Polygon	绘制封闭多边形
PolyLine	绘制多边线条
PolylineTo	以当前画笔所在位置绘制多边线条
PolyPolygon	绘制多个封闭多边形
PolyPolyline	绘制多个多边线条

以上几个函数的使用方法大同小异，以 Polygon()函数来做说明。

BOOL Polygon(HDC hdc,CONST POINT 点数组指针,int 多边形点数); //绘制多边形

第 2 个参数输入的是一个 POINT 结构的数组指针，POINT 可用于描述一个坐标点，其结构如下。

```
typedef struct tagPOINT {
    LONG x;           // X坐标
    LONG y;           // Y坐标
} POINT;
```

下面我们就利用以上函数来做一个多边形绘制的范例。

## » 范例 ch2\_3

展示多边形绘制函数的用法，并在窗口中绘制多边形。

程序代码: InitInstance()

```
1  //****初始函数*****
2  //1.各多边形顶点数组初始化
3  //2.建立画笔与画刷
4  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
5  {
6      HWND hWnd;
7      HDC hdc;
8      int i;
9      const double pi = 3.1415926535;
10
11     hInst = hInstance;
12
13     hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
14         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
15
16     if (!hWnd)
17     {
18         return FALSE;
```

```
19 }
20
21 MoveWindow(hWnd, 10, 10, 600, 450, true);
22 ShowWindow(hWnd, nCmdShow);
23 UpdateWindow(hWnd);
24
25 for(i=0; i<=4; i++)
26 {
27     poly1[i].x = 100 + 100 * sin(i*72*pi/180);
28     poly1[i].y = 100 + 100 * cos(i*72*pi/180);
29
30     poly2[i].x = poly1[i].x + 300;
31     poly2[i].y = poly1[i].y;
32
33     poly3[i].x = poly1[i].x + 180;
34     poly3[i].y = poly1[i].y + 200;
35 }
36
37 hPen = CreatePen(PS_SOLID, 5, RGB(255, 0, 0));
38 hBru = CreateHatchBrush(HS_BDIAGONAL, RGB(0, 255, 0));
39
40 hdc = GetDC(hWnd);
41 MyPaint(hdc);
42 ReleaseDC(hWnd, hdc);
43
44 return TRUE;
45 }
```

**程序说明**

- (1) 第 25~35 行：将 3 个声明为全局变量的顶点数组“poly1[]”、“poly2[]”和“poly3[]”初始化。
- (2) 第 37~38 行：建立画笔与画刷。
- (3) 第 41 行：调用 MyPaint() 函数绘图。

**程序代码：MyPaint()**

```
1  //****自定义绘图函数*****
2  // 绘制多边形
3  void MyPaint(HDC hdc)
4  {
5      SelectObject(hdc, hPen);
6      SelectObject(hdc, hBru);
7      PolylineTo(hdc, poly1, 5);
8      Polyline(hdc, poly2, 5);
9      Polygon(hdc, poly3, 5);
10 }
```

**程序说明**

- (1) 第 5~6 行：选用自定义画笔及画刷。
- (2) 第 7~10 行：多边形绘图函数分别按照顶点数组“poly1[]”、“poly2[]”和“poly3[]”绘图。

## 运行结果

程序运行结果如图 2-8 所示。

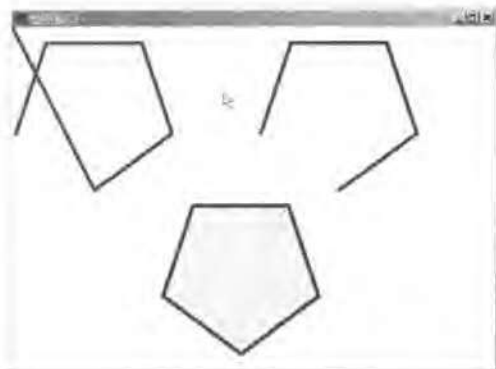


图 2-8

## 3. 封闭图形函数

下边再来讨论一下封闭几何图形的绘制方法，上一小节中介绍了画矩形的 `Rectangle()` 函数，现在来看看画椭圆形的 GDI 函数。

```

BOOL Ellipse( HDC hdc,                //绘制椭圆形
              int 外围矩形左上点 x 坐标,
              int 外围矩形左上点 y 坐标,
              int 外围矩形右下点 x 坐标,
              int 外围矩形右下点 y 坐标 );
    
```

上面的函数说明确定一个外围矩形就可以产生椭圆形，如图 2-9 所示。

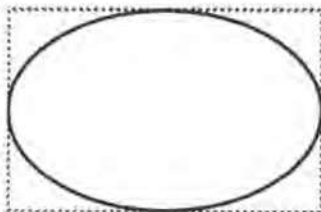


图 2-9

图 2-9 中的虚线部分就是外围矩形，当调用 `Ellipse()` 函数时，只要输入外围矩形的坐标值，就会画出实线部分的椭圆形。

下面再来看看绘制圆角矩形的函数。

```

BOOL RoundRect( HDC hdc,                //绘制圆角矩形
                int 外围矩形左上点 x 坐标,
                int 外围矩形左上点 y 坐标,
                int 外围矩形右下点 x 坐标,
                int 外围矩形右下点 y 坐标,
                int 圆角上椭圆长,
                int 圆角上椭圆高 );
    
```

在这个函数中，除了给定外围矩形的坐标外，还给出了圆角上椭圆的长与高，用来表示圆角的

弧度, 将以上想法以图 2-10 表示, 实线部分是实际画出的圆角矩形。

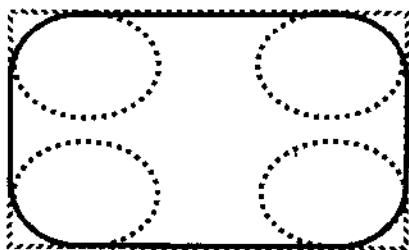


图 2-10

最后来说明绘制扇形 (Pie) 与弓形 (Chord) 的函数, 扇形与弓形都是椭圆的一部分, 它们的区别如图 2-11 和图 2-12 所示。

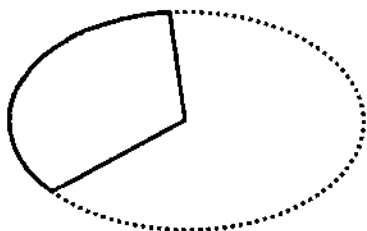


图 2-11

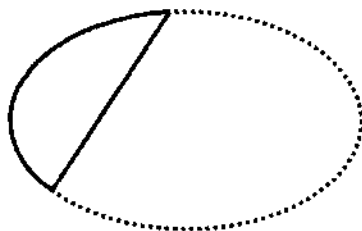


图 2-12

扇形与弓形都有连接的起点与终点, 不同之处在于, 扇形还会与椭圆的中心点相连接, 而弓形则直接连接起点与终点。绘制扇形的函数为 Pie(), 绘制弓形的函数则为 Chord(), 两者输入的参数意义都相同, 下面以 Pie() 函数为例来进行说明。

```

BOOL Pie(          HDC hdc,          //绘制扇形
    int 外围矩形左上点 x 坐标,
    int 外围矩形左上点 y 坐标,
    int 外围矩形右下点 x 坐标,
    int 外围矩形右下点 y 坐标,
    int 起点 x 坐标,
    int 起点 y 坐标,
    int 终点 x 坐标,
    int 终点 y 坐标 );
  
```

接下来通过一个范例来看看这几个绘图函数的实际使用方法。

## » 范例 ch2\_4

显示封闭图形绘制函数的用法, 并在窗口中绘制不同图形。

程序代码: InitInstance()

```

1  //****初始化函数*****
2  // 建立画笔与画刷
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {
5      HWND hWnd;
6      HDC hdc;
  
```



```

7   int i;
8
9   hInst = hInstance;
10
11  hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
12      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
13
14  if (!hWnd)
15  {
16      return FALSE;
17  }
18
19  MoveWindow(hWnd, 10, 10, 600, 450, true);
20  ShowWindow(hWnd, nCmdShow);
21  UpdateWindow(hWnd);
22
23  hPen = CreatePen(PS_SOLID, 3, RGB(255, 0, 0));
24  for(i=0; i<=3; i++)
25  {
26      hBru[i] = CreateHatchBrush(sBru[i], RGB(0, 0, 255));
27  }
28
29  hdc = GetDC(hWnd);
30  MyPaint(hdc);
31  ReleaseDC(hWnd, hdc);
32
33  return TRUE;
34 )

```

### 程序说明

- (1) 第 23 行：建立宽度为 3 的蓝色画笔。
- (2) 第 24~27 行：建立 4 种阴影画刷，画笔、画刷及画刷类型都在全局变量区中声明。

### 程序代码：MyPaint()

```

1   //****自定义绘图函数*****
2   void MyPaint(HDC hdc)
3   {
4       SelectObject(hdc, hPen);
5
6       SelectObject(hdc, hBru[0]);
7       Ellipse(hdc, 20, 20, 270, 150);           //画椭圆形
8       TextOut(hdc, 120, 160, "椭圆形", strlen("椭圆形"));
9
10      SelectObject(hdc, hBru[1]);
11      RoundRect(hdc, 300, 20, 550, 150, 30, 30); //画圆角矩形
12      TextOut(hdc, 400, 160, "圆角矩形", strlen("圆角矩形"));
13
14      SelectObject(hdc, hBru[2]);
15      Pie(hdc, 20, 210, 270, 340, 50, 50, 300, 300); //画扇形

```

```
16 TextOut(hdc,120,350,"扇形",strlen("扇形"));
17
18 SelectObject(hdc,hBru[3]);
19 Chord(hdc,300,210,550,340,50,50,600,300); //画弓形
20 TextOut(hdc,400,350,"弓形",strlen("弓形"));
21 }
```

#### 程序说明

在 MyPaint()函数中,以各种不同的画刷样式填充绘制各种前面介绍过的封闭图形,并输出说明文字。

#### 运行结果

程序运行结果如图 2-13 所示。

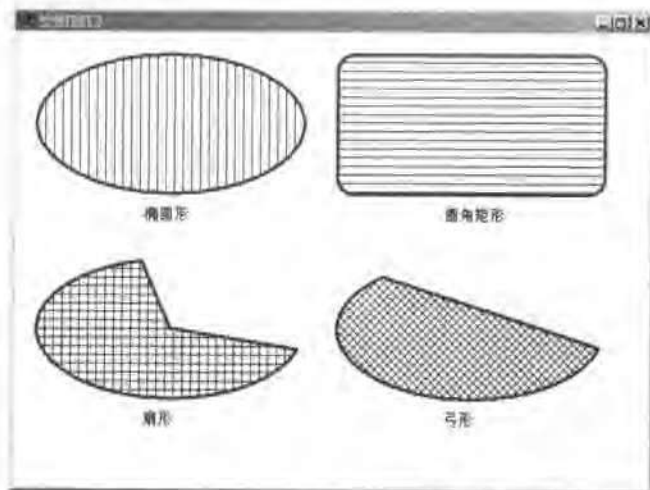


图 2-13

到目前为止,就使用 GDI 在窗口中绘图的方式做了一些简单的介绍,想必您已经对窗口绘图的流程有了一些清楚的概念。下一小节将介绍窗口中绘制位图的方法。当然,它还是属于 GDI 范畴的。

### 2.1.4 绘制位图

位图是属于 GDI 的对象之一,在一套游戏开发过程中,常常需要运用大量的位图来构建游戏的所有画面。这一小节里将介绍在窗口中绘制位图的方式和基本的贴图技巧。

以游戏程序来说,由于使用的位图数量相当多,因此都会先将位图存成文件,等到程序需要时再将文件加载到窗口中。将位图从文件中加载到绘制窗口中必须经过以下几个步骤。

- (1) 从文件中加载位图 (BITMAP) 对象。
- (2) 建立一个与窗口 DC 兼容的内存 DC。
- (3) 内存 DC 使用步骤 1 所建立的位图对象。
- (4) 将内存 DC 的内容粘贴到窗口 DC 中,完成显像的操作。

以上就是大致的流程,接下来一步步地说明如何完成这些操作。

#### 步骤一: 加载位图

要从文件加载位图,常使用 LoadImage()函数。

```
HANDLE LoadImage(          HINSTANCE 来源实体,          //加载位图
                        LPCTSTR 名称,
                        UINT 位图类型,
                        int 加载宽度,
                        int 加载高度,
                        UINT 加载方式 );
```

表 2-2 针对这个函数的各个参数进行了更详细的说明。

表 2-2

参 数	说 明
HINSTANCE 来源实体	包含位图所在的实体,若要加载的位图存在于硬盘或者资源文件中,则将此参数设为“NULL”
LPCTSTR 名称	要加载位图所在的路径与文件名或者资源名称
UINT 位图类型	加载位图的类型,有下列3种: ◆ IMAGE_BITMAP: 加载的位图为一般图文件,扩展名为“.bmp” ◆ IMAGE_CURSOR: 加载的位图为光标图标,扩展名为“.cur” ◆ IMAGE_ICON: 加载的位图为图标,扩展名为“.ico”
int 加载宽度	位图加载的宽度,单位为像素
int 加载高度	位图加载的高度,单位为像素
UINT 加载方式	设定位图的加载方式,若是从文件中加载位图,则设为“LR_LOADFROMFILE”

## 步骤二：建立与窗口 DC 兼容的内存 DC

内存 DC 并不是真正设备的 DC,在这里把它解释为一个缓冲区或许会更恰当些。内存 DC 用来暂存加载的位图,由于最终会把存储在这个内存 DC 上的位图贴到真正窗口的 DC 上,因此这个内存 DC 必须跟窗口 DC 的性质兼容。可调用 CreateCompatibleDC()函数来建立内存 DC。

```
HDC CreateCompatibleDC( HDC hdc );          //建立兼容 DC
```

函数中输入的惟一参数就是要与内存 DC 兼容的目的 DC。

跟窗口 DC 一样,内存 DC 使用后也必须进行释放的操作,释放内存 DC 所调用的函数为 DeleteDC()。

```
DeleteDC(HDC DC 名称);          //释放 DC
```

## 步骤三：选用位图对象

位图对象是 GDI 的 6 种对象之一,内存 DC 选用位图对象的方法和前面介绍的选用画笔或画刷的方式相同,都是通过调用 SelectObject()函数来实现。

## 步骤四：贴图

把内存 DC 中的位图复制到显示的 DC 上,这个操作被称为“贴图”。这个操作所使用的函数是 BitBlt(),它相当重要,从现在开始到本书结束将会一直用到它,这个函数的内容如下。

```
BOOL BitBlt(          HDC 目的DC,          //贴图
            int 目的DC X坐标,
            int 目的DC Y坐标,
            int 贴到目的DC的宽度,
            int 贴到目的DC的高度,
            HDC 来源DC,
            int 来源DC X坐标,
            int 来源DC Y坐标,
```

DWORD 贴图方式);

下面利用实际例子来说明 BitBlt()函数中几个参数所代表的意义。假设现在已经建立了一个内存 DC, 名为“mdc”, 其中已加载了所要显示的位图, 而窗口的 DC 名为“hdc”, 贴图程序代码如下。

```
BitBlt(hdc, 50, 50, 350, 250, mdc, 200, 100, SRCCOPY);
```

这行程序表示, 从来源 DC(mdc)坐标点 (200,100) 的地方开始向右向下剪裁出宽 350、高 250 的区域, 并将其贴到目的 DC(hdc)以坐标点 (50,50) 为原点的区域中, 如图 2-14 和图 2-15 所示。

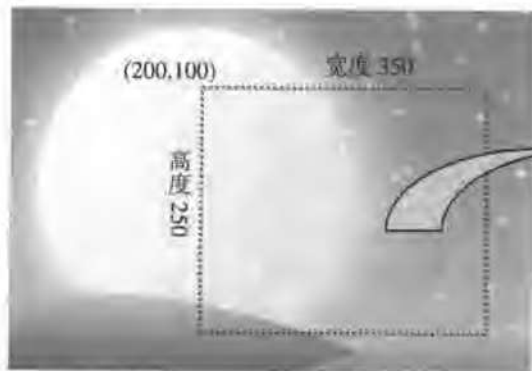


图 2-14

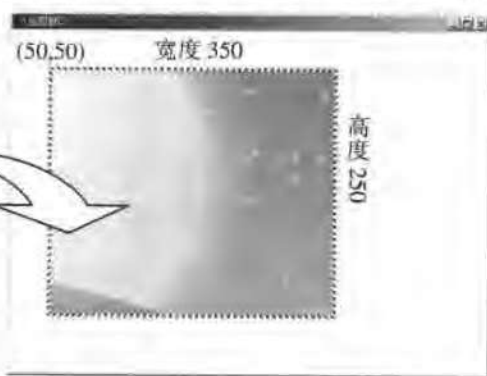


图 2-15

这样, 位图就会出现在显示窗口中。下面的这个范例, 便是将以上所说明的各个贴图步骤转化为实际的程序代码, 并在窗口中绘制一张位图。

## » 范例 ch2\_5

从文件中加载位图, 并显示在窗口上。

**程序代码: 全局变量声明**

```
1 //全局变量声明
2 HINSTANCE hInst;
3 HBITMAP hbmp;
4 HDC mdc;
```

**程序说明**

- (1) 第 3 行: 声明一个位图对象“hbmp”。
- (2) 第 4 行: 声明一个内存 DC(mdc), 用来暂存位图。

**程序代码: InitInstance()**

```
1 //****初始化函数*****
2 // 1.建立与窗口 DC 兼容的内存 DC
3 // 2.从文件加载位图并存至内存 DC 中
4 BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
5 {
6     HWND hwnd;
7     HDC hdc;
8 }
```

```

9  hInst = hInstance;
10
11  hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
12      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
13
14  if (!hWnd)
15  {
16      return FALSE;
17  }
18
19  MoveWindow(hWnd, 10, 10, 600, 450, true);
20  ShowWindow(hWnd, nCmdShow);
21  UpdateWindow(hWnd);
22
23  hdc = GetDC(hWnd);
24  mdc = CreateCompatibleDC(hdc);
25
26  hbm = (HBITMAP)LoadImage(NULL, "bg.bmp", IMAGE_BITMAP, 600, 450,
27      LR_LOADFROMFILE);
28  SelectObject(mdc, hbm);
29  MyPaint(hdc);
30  ReleaseDC(hWnd, hdc);
31
32  return TRUE;
33 }

```

## 程序说明

- (1) 第 23~24 行：先获取窗口 DC(hdc)，再建立一个与 hdc 兼容的内存 DC(mdc)。
- (2) 第 26 行：加载文件名为“bg.bmp”的位图到“hbm”。
- (3) 第 27 行：将“hbm”存到 mdc 中。
- (4) 第 29 行：调用 MyPaint()函数绘图。

## 程序代码：MyPaint()

```

1  //****自定义绘图函数****
2  void MyPaint(HDC hdc)
3  {
4      BitBlt(hdc, 0, 0, 600, 450, mdc, 0, 0, SRCCOPY); //贴图
5  }

```

## 程序说明

以上程序代码说明，在每次调用 MyPaint()函数时，都使用 BitBlt()函数将 mdc 中的内容贴到窗口 DC 中，贴图的大小为 600×450 像素。

## 程序代码：WndProc()

```

1  //****消息处理函数****
2  LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
3  lParam)
4  {

```

```
4  PAINTSTRUCT ps;  
5  HDC hdc;  
6  
7  switch (message)  
8  {  
9      case WM_PAINT:                //窗口重绘消息  
10     hdc = BeginPaint(hWnd, &ps);  
11     MyPaint(hdc);  
12     EndPaint(hWnd, &ps);  
13     break;  
14     case WM_DESTROY:              //窗口结束消息  
15     DeleteDC(mdc);  
16     DeleteObject(hbmp);  
17     PostQuitMessage(0);  
18     break;  
19     default:                      //其他消息  
20     return DefWindowProc(hWnd, message, wParam, lParam);  
21 }  
22 return 0;  
23 }
```

#### 程序说明

(1) 第 15 行：由于已将内存 DC(mdc)声明为全局变量，因此在程序结束时必须调用 DeleteDC() 函数将其释放掉。

(2) 第 16 行：删除位图对象。

#### 运行结果

程序运行结果如图 2-16 所示。



图 2-16

#### Raster 运算

BitBlt()函数最后一个参数所输入的是称为“Raster”的运算值，这个值是用来设定内存 DC 到目的 DC 的贴图方式。范例中输入的 Raster 值是 SRCCOPY，表示贴图后的位图与原来的位图完全

一样。表 2-3 中列出了可使用的 Raster 值及说明。

表 2-3

Raster 值	说 明
BLACKNESS	将来源位图转换为黑色
DSTINVERT	将目的地 DC 做“NOT”运算
MERGECOPY	将选择的笔刷与来源位图做“AND”运算
MERGEPAINT	先将来源位图做“NOT”运算，再与目的地 DC 做“OR”运算
NOTSRCCOPY	将来源位图做“NOT”运算
NOTSRCERASE	先将来源位图与目的地 DC 做“OR”运算，再将其做“NOT”运算
PATCOPY	将选择的笔刷贴到目的地 DC 上
PATINVERT	将目的地 DC 与选择的笔刷做“XOR”运算
PATPAINT	先将来源位图做“NOT”运算，再与笔刷做“OR”运算，最后再与目的地 DC 做“OR”运算
SRCAND	将来源位图与目的地 DC 做“AND”运算
SRCCOPY	将来源位图贴到目的地 DC 上
SRCERASE	先将目的地 DC 做“NOT”运算，再与来源位图做“AND”运算
SRCINVERT	将来源位图与目的地 DC 做“XOR”运算
SRCPAINT	将来源位图与目的地 DC 做“OR”运算
WHITENESS	将来源位图转换为白色

对于使用 Raster 值的特殊贴图应用，将在下一节制作透明效果时会遇到。

## 2.2 游戏画面特效制作

在前面内容里面，对于屏幕绘图的基本概念与技巧已经介绍得差不多了，在这一章后半部分的内容里，要讲解的则是一般设计 2D 游戏画面时经常会使用到的绘图特效。这一节将介绍“透明”与“半透明”效果的制作方法。

### 2.2.1 透明效果

由于所有的图文件都是以一个四方矩形来存储的，但有时我们可能会需要把一张怪物图片贴到窗口的背景图上，而在这种情况下如果我们直接进行贴图，其结果如图 2-17 所示。



图 2-17

这似乎不是所要的结果。这一小节里所要讨论的透明效果，就是要利用 BitBlt() 贴图函数以及 Raster 值的运算来将图片中不必要的部分去掉（又称去背），使得图中的主题可以与背景图完全融合。

制作透明效果有多种方法，但基本上都是利用贴图时不同的 Raster 运算，通过转换而最后产生相同的透明效果。在这里先来介绍一种透明运算的方法。

以图 2-17 中的恐龙图为例，首先必须准备一张位图，它的色彩分配如图 2-18 所示。



图 2-18

图中左边的图是要去背并贴到背景上的前景图，右边的黑白图称为“屏蔽图”，在透明的过程中会用到它。把要去背的位图与屏蔽图合并成同一张图，透明的时候再按照需要来进行裁切。可以把它分成两张图，但这样程序必须运行两次图文件加载的操作。

有了屏蔽图就可以利用贴图函数来产生透明效果了，所需进行的贴图步骤如下。

- (1) 将屏蔽图与背景图做“AND”（Raster 值为 SRCAND）运算，贴到目的地 DC 中。
- (2) 将前景图与背景图做“OR”（Raster 值为 SRCPAINT）运算，贴到目的地 DC 中。

为什么经过上面的两个步骤就能产生透明的效果呢？可以参看图 2-19。

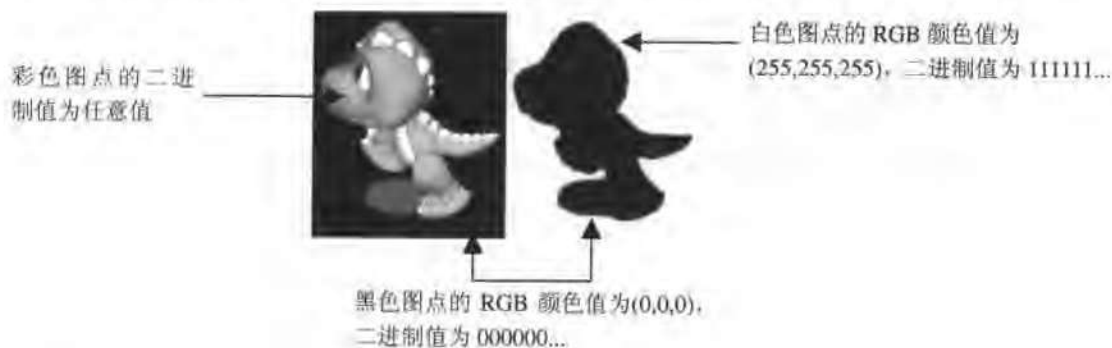


图 2-19

接下来说明上述两个步骤所产生的图点色彩的变化。

### 1. 屏蔽图与背景图做“AND”运算

- (1) 屏蔽图中的黑色部分与背景图做“AND”运算：

$$\begin{array}{rcl}
 000000... & \text{———} & \text{屏蔽图中黑色图点的颜色值} \\
 \text{AND) } 011010... & \text{———} & \text{背景图中彩色图点的颜色值} \\
 \hline
 000000... & \text{———} & \text{运算后变成黑色}
 \end{array}$$

- (2) 屏蔽图中的白色部分与背景图做“AND”运算：



```

111111...  ————— 屏蔽图中白色图点的颜色值
AND) 101010...  ————— 背景图中彩色图点的颜色值
—————
101010...  ————— 运算后还是原来背景图的色彩

```

经过这一运算所产生的结果如图 2-20 所示。



图 2-20

## 2. 前景图与背景图做“OR”运算

(1) 前景图中的彩色部分与图 2-20 做“OR”运算：

```

101011...  ————— 前景图中彩色图点的颜色值
OR) 000000...  ————— 背景图中变成黑色的图点颜色值
—————
101011...  ————— 运算后变成前景图的色彩

```

(2) 前景图中的黑色部分与图 2-20 做“OR”运算：

```

000000...  ————— 前景图中黑色图点的颜色值
OR) 101010...  ————— 背景图中彩色图点的颜色值
—————
101010...  ————— 运算后还是原来背景图的色彩

```

经过这一运算后所显示的画面就是所需的透明图了，如图 2-21 所示。



图 2-21

最后来看看产生这个透明效果的范例的程序内容。

## » 范例 ch2\_6

透明效果的显示。

**程序代码：全局变量声明**

```

1  //全局变量声明
2  HINSTANCE  hInst;
3  HBITMAP    bg,dra;
4  HDC        mdc;

```

**程序说明**

- (1) 第3行：声明两个位图对象“bg”与“dra”，分别用来存储背景图与前景的恐龙图。  
 (2) 第4行：声明一内存 DC “mdc”，用来暂存位图。

**程序代码：InitInstance()**

```

1  //****初始化函数*****
2  // 1.建立与窗口 DC 兼容的内存 DC
3  // 2.从文件中加载背景图与恐龙图
4  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
5  {
6      HWND hWnd;
7      HDC hdc;
8
9      hInst = hInstance;
10
11     hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
12         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
13
14     if (!hWnd)
15     {
16         return FALSE;
17     }
18
19     MoveWindow(hWnd,10,10,600,450,true);
20     ShowWindow(hWnd, nCmdShow);
21     UpdateWindow(hWnd);
22
23     hdc = GetDC(hWnd);
24     mdc = CreateCompatibleDC(hdc);
25
26     bg = (HBITMAP)LoadImage(NULL,"bg.bmp",IMAGE_BITMAP,600,450,LR_LOADFROMFILE);
27     dra = (HBITMAP)LoadImage(NULL,"dra.bmp",IMAGE_BITMAP,170,99,LR_LOADFROMFILE);
28
29     MyPaint(hdc);
30     ReleaseDC(hWnd,hdc);
31
32     return TRUE;
33 }

```

**程序说明**

- (1) 第23~24行：获得窗口 DC，并建立与其兼容的内存 DC(mdc)。  
 (2) 第26~27行：分别从文件中加载背景图与恐龙图到“bg”与“dra”中。

### 程序代码: MyPaint()

```
1  //****自定义绘图函数*****
2  // 透明贴图
3  void MyPaint(HDC hdc)
4  {
5      SelectObject(mdc,bg);
6      BitBlt(hdc,0,0,600,450,mdc,0,0,SRCCOPY);
7
8      SelectObject(mdc,dra);
9      BitBlt(hdc,280,320,85,99,mdc,85,0,SRCAUD);
10     BitBlt(hdc,280,320,85,99,mdc,0,0,SRCPAINT);
11 }
```

### 程序说明

(1) 第5~6行: 先将背景图贴到显示窗口中。

(2) 第8行: 贴完背景图后, 选用恐龙图到“mdc”中。

(3) 第9行: 进行制作透明贴图的第一步骤, 即将屏蔽图与背景图做“AND”运算。屏蔽图在整张恐龙图中, 最左上角起始位置点的坐标为(85,0), BitBlt()函数最后一个 Raster 参数值设为 SRCAND。

(4) 第10行: 进行制作透明贴图的第二步骤, 即将前景图与背景图做“OR”运算。前景图在整张恐龙图中, 最左上角起始位置点的坐标为(0,0), BitBlt()函数最后一个 Raster 参数值设为 SRCPAINT。

### 运行结果

程序运行结果如图 2-22 所示。



图 2-22

通过 BitBlt()贴图函数及 Raster 运算值的设定, 很简单地就做出了所要的透明效果, 这种方法在设计 2D 游戏的一些画面内容时使用相当频繁。

## 2.2.2 半透明效果

半透明在游戏中通常用来呈现若隐若现的特殊效果。事实上这种效果的运用相当频繁, 比如薄雾、鬼魂或隐形人物等。有时会以半透明的手法来表现。这一小节就来介绍半透明效果的制作方法。

图 2-23 是将一张位图经过半透明处理后显示在背景上的效果。



图 2-23

### 1. 半透明的制作原理

简单地说，半透明效果就是前景图案与背景图案像素颜色的混合。从图 2-23 中观察半透明效果呈现的区域，可以看到背景图案，也可看到前景的人物图案。什么是前景图案与背景图案像素颜色的混合呢？这就要从位图的基本结构开始谈起了。

一张位图是由许多的像素所组成的，每一个像素中都包含红（R）、绿（G）、蓝（B）三原色的色彩值，由这三种原色值来决定该像素的色彩。而要呈现半透明效果，必须将前景图与背景图彼此对应像素的颜色依某一比例来进行调配，这个比例就叫做“不透明度”。

以没有进行半透明处理，单纯地将一张前景图贴到背景图上的一块区域来说，前景图的不透明度是 100%，而背景图在这一块区域上的不透明度则是 0%（完全透明，所以看不见背景），也就是说在这块区域上，背景图的色彩完全派不上用场。

可是如果想要有半透明的效果，让前景图看起来稍微透明一点，那就需要确定不透明度的值。假设确定不透明度是 70%，也就是说前景图像素颜色在显示位置上的比例是 70%，而剩下的 30%当然就是背景像素的颜色了。

这样，如果可以将要显示区域内的每一个像素颜色按照一定的不透明度比例进行合成，那么最后整个区域所呈现出来的就是所要的半透明效果了。综合上面的说明，可以整理出一个简单的公式如下：

$$\text{半透明图色彩} = \text{前景图色彩} \times \text{不透明度} + \text{背景图色彩} \times (1 - \text{不透明度})$$

### 2. 半透明的操作步骤

清楚了半透明制作的原理后，接下来说明程序产生半透明效果的实际步骤。

#### 步骤一：取得位图结构

位图结构包含了一些位图的基本信息，由于我们在制作半透明效果时会用到，因此在从文件加载位图后，必须先取得该位图的结构，而取得位图结构的函数如下。

```
int GetObject( HGDIOBJ    GDI 对象,           //取得 GDI 对象结构
               int         结构大小,
               LPVOID      结构变量 );
```

上面这个函数用于取得 GDI 对象的信息，包含这里所谈的位图，其中第 3 个参数是一个结构变量，如果是用于取得位图的信息，则输入一个位图结构的地址，而 Windows API 中所定义的位图结构 (BITMAP) 如下。

```
typedef struct tagBITMAP {
    LONG    bmType;           //位图类型，必须设为 0
    LONG    bmWidth;          //位图宽度
    LONG    bmHeight;         //位图长度
    LONG    bmWidthBytes;     //每一列像素所占 Byte 数
    WORD    bmPlanes;         //颜色平面数
    WORD    bmBitsPixel;      //像素的位数
    LPVOID   bmBits;          //位图内存指针
} BITMAP;
```

后面将会用到 bmWidth、bmHeight、bmWidthBytes 及 bmBitsPixel 这几个结构成员的信息。

在此举个例子来说明取得位图结构的方法，假设现在有一个位图名称为“bitmap”，位图结构变量名称为“bm”，则使用 GetObject() 函数取得 BITMAP 结构的程序代码如下。

```
GetObject(bitmap, sizeof(BITMAP), &bm);
```

这样，位图结构 bm 中的各个结构成员便包含了位图 bitmap 的基本信息。

## 步骤二：建立暂存数组

取得了位图的结构，接下来必须先建立一个暂存数组准备存储位图中所有像素的颜色值。这个暂存数组的大小是由前一个步骤中所取得位图的 bmHeight 与 bmWidthBytes 信息来决定的，因此，必须利用指针来动态建立。延续前一个例子，若要建立一个可存储 bitmap 所有像素颜色值的暂存数组，程序代码如下。

```
unsigned char *px = new unsigned char [bm.bmHeight * bm.bmWidthBytes];
```

这里，因为 unsigned char 变量类型大小是 1Byte，所以这个数组的每个元素大小也就是 1Byte (8bits)。以一张 24bits 色彩的位图来说，它的每个像素是以 24bits 来表示颜色的，其中 B (蓝)、G (绿)、R (红) 三原色各占 8 个 bits。

因此，在下面的步骤中，当取出位图的所有颜色并存储在这个数组中时，每一个像素会占用 3 个数组元素来存储 B、G、R 的颜色值。

## 步骤三：取得位图位值

建立了暂存数组之后，要取出位图的所有颜色值存储到数组中就简单多了，有一个现成的 API 函数可以使用。

```
LONG GetBitmapBits( HBITMAP    位图,           //取得位图位值
                    LONG        要取得的 Byte 数,
                    LPVOID       存储的数组指针 );
```

使用此函数取得位图位值的程序代码如下。

```
GetBitmapBits(bitmap, bm.bmHeight * bm.bmWidthBytes, px);
```

下面以图示来说明像素颜色值存储在数组中的对应关系，如图 2-24 所示。

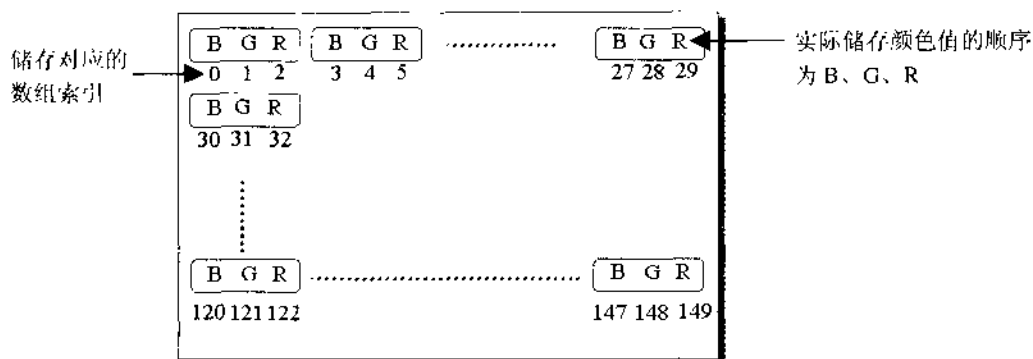


图 2-24

#### 步骤四：合成像素颜色值

取得了位图的所有像素颜色值之后，接下来的工作就是按照不透明度来设定半透明区域内每个像素的颜色了。

此时应该会有两个像素颜色数组，一个是前景图的，一个则是背景图的。根据实际要显示半透明区域的坐标，将这两个数组算出对应的元素进行前面讲过的颜色合成运算，再存回暂存数组中，那么数组中所存储的便是已经完成半透明的颜色值了。这个步骤的实际处理过程，将在后面范例中做详细说明。

#### 步骤五：重设位图颜色

处理完暂存数组中半透明的颜色值之后，最后一个操作就是根据数组的内容来重设位图的颜色。这个操作同样可以使用一个 API 的函数来完成。

```
LONG SetBitmapBits(    HBITMAP 位图,           //设定位图位值
                      DWORD 颜色数组大小,
                      CONST VOID 数组指针 );
```

以上的 5 个步骤都完成之后，所要的半透明图也就完成了，剩下的就只有贴图操作了。

接下来通过下面的范例，可以帮助您更加熟悉半透明效果的制作方法。

### 》范例 ch2\_7

取得前景图与背景图的颜色值，以前景图的不透明度 30% 和背景图的不透明度 70% 进行半透明处理，制作半透明效果。

#### 程序代码：全局变量声明与常数定义

```
1 //全局变量声明
2 HINSTANCE hInst;
3 HBITMAP bg, girl;
4 HDC hdc;
5
6 //常数定义
7 const int xstart = 50;
8 const int ystart = 20;
```

#### 程序说明

第 7~8 行：定义两常数“xstart”与“ystart”，分别是半透明图贴图的起始坐标。

## 程序代码: InitInstance()

```

1  //****初始函数*****
2  // 半透明处理, 处理后将半透明图存回girl 中
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {
5      HWND hWnd;
6      HDC hdc;
7
8      hInst = hInstance;
9
10     hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
11         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
12
13     if (!hWnd)
14     {
15         return FALSE;
16     }
17
18     MoveWindow(hWnd, 10, 10, 600, 450, true);
19     ShowWindow(hWnd, nCmdShow);
20     UpdateWindow(hWnd);
21
22     hdc = GetDC(hWnd);
23     mdc = CreateCompatibleDC(hdc);
24
25     BITMAP bm1, bm2;
26     unsigned char *px1, *px2;
27
28     //处理背景图
29     bg = (HBITMAP)LoadImage(NULL, "bg.bmp", IMAGE_BITMAP, 600, 450, LR_LOADFROMFILE);
30     GetObject(bg, sizeof(BITMAP), &bm1);
31
32     if(bm1.bmBitsPixel != 32 && bm1.bmBitsPixel != 24)
33     {
34         MessageBox(NULL, "此程序只能在 32 bit 或 24 bit 显示模式中运行", "警告", 0);
35         return FALSE;
36     }
37
38     px1 = new unsigned char [bm1.bmHeight * bm1.bmWidthBytes];
39     GetBitmapBits(bg, bm1.bmHeight * bm1.bmWidthBytes, px1);
40
41     //处理前景图
42     girl = (HBITMAP)LoadImage(NULL, "girl.bmp", IMAGE_BITMAP, 298, 329,
        LR_LOADFROMFILE);
43     GetObject(girl, sizeof(BITMAP), &bm2);
44     px2 = new unsigned char [bm2.bmHeight * bm2.bmWidthBytes];
45     GetBitmapBits(girl, bm2.bmHeight * bm2.bmWidthBytes, px2);
46

```

```
47 int xend,yend;
48 int x,y,i;           //循环变量
49 int rgb_b;
50 int PxBytes = bml.bmBitsPixel / 8 ;
51
52 xend = xstart + 298;
53 yend = ystart + 329;
54
55 //处理背景图像像素颜色
56 for(y=ystart;y<yend;y++)
57 {
58     for(x=xstart;x<xend;x++)
59     {
60         rgb_b = y * bml.bmWidthBytes + x * PxBytes ;
61
62         px1[rgb_b] = px1[rgb_b] * 0.7;
63         px1[rgb_b+1] = px1[rgb_b+1] * 0.7;
64         px1[rgb_b+2] = px1[rgb_b+2] * 0.7;
65     }
66 }
67
68 //处理前景图像像素颜色
69 for(y=0;y<(bm2.bmHeight); y++)
70 {
71     for(x=0;x<bm2.bmWidth; x++)
72     {
73         rgb_b = y * bm2.bmWidthBytes + x * PxBytes ;
74         i = (ystart+y) * bml.bmWidthBytes + (xstart+x) * PxBytes;
75
76         px2[rgb_b] = px2[rgb_b] *0.3 + px1[i];
77         px2[rgb_b+1] = px2[rgb_b+1] *0.3 + px1[i+1];
78         px2[rgb_b+2] = px2[rgb_b+2] *0.3 + px1[i+2];
79     }
80 }
81
82 SetBitmapBits(girl,bm2.bmHeight*bm2.bmWidthBytes,px2);
83
84 MyPaint(hdc);
85
86 ReleaseDC(hWnd,hdc);
87 delete [] px1;
88 delete [] px2;
89
90 return TRUE;
91 }
```

#### 程序说明

在初始函数中主要对前景图 (girl) 与背景图 (bg) 进行处理, 并将完成后的半透明图存回 girl 中。以后在窗口中显示时, 只需贴上 girl 的内容而不必重新进行半透明处理。



- (1) 第 25 行: 声明两个 BITMAP 结构“bm1”和“bm2”分别用来存储位图 bg 与 girl 的信息。
- (2) 第 26 行: 声明两指针“px1”与“px2”分别用来指向存储位图色彩值的数组。
- (3) 第 29~30 行: 加载背景图并取得背景图信息, 存于 bm1 中。
- (4) 第 32~36 行: 根据 bm1.bmBitsPixel 的信息判断目前的显示系统, bm1.bmBitsPixel 是背景图的显示位信息, 此值会依显示系统不同而改变。由于使用的是 24bits 的位图, 在 16bits 以下的显示系统中, 像素的原色值所占的 bit 数会自动被系统改变, 因此无法以 1Byte 来存储, 使程序无法正常运行。
- (5) 第 38~39 行: 建立暂存数组“px1”, 并将取得的背景图像素色彩值存入 px1 中。
- (6) 第 41~45 行: 依次进行加载前景图、取得前景图信息、建立暂存数组、取得像素色彩值并存入暂存数组 px2 中的操作。
- (7) 第 50 行: 计算目前显示系统中每一个像素所占用的 Byte 数 (每一个 Byte 在暂存数组中即为一个元素), 有助于后面求出像素各原色值存储的元素索引值。
- (8) 第 52~53 行: 求出要显示半透明区域的右下角点的坐标。
- (9) 第 56、58 行: 以半透明贴图区域的左上角点 (xstart, ystart) 和右下角点 (xend, yend) 的坐标值作为循环的初始值与临界值, 仅取出背景图贴图区域像素的颜色值做处理。
- (10) 第 60 行: 计算半透明贴图区域内每个像素第一个颜色值 (蓝色) 所在的元素索引。
- (11) 第 62~64 行: 依次将像素的 B、G、R 色彩值乘以背景图的不透明度 70%。
- (12) 第 69、71 行: 前景图要处理全部像素的颜色, 所以循环的起始值为 0, 临界值分别为图的高 (bm2.Height) 与宽 (bm2.Width)。
- (13) 第 73 行: 计算前景图每个像素第一个颜色值 (蓝色) 的所在的元素索引。
- (14) 第 74 行: 计算前景图贴图时对应背景区域上像素第一个颜色值 (蓝色) 在“px1”数组中的元素索引。
- (15) 第 76~78 行: 套用半透明色彩合成的公式, 依次将像素的 B、G、R 色彩值乘以前景图的不透明度 30%, 并与“px1”中已处理过的背景图色彩值相加, 然后回存到“px2”数组中便完成了半透明色彩的合成。
- (16) 第 82 行: 以数组“px2”的内容重设前景图 girl 的颜色, girl 即为已完成的半透明图。
- (17) 第 84 行: 调用 MyPaint() 函数进行第一次窗口贴图。
- (18) 第 86~88 行: 释放 DC 及内存。

## 程序代码: MyPaint()

```

1  //****自定义绘图函数*****
2  void MyPaint (HDC hdc)
3  {
4      //贴上背景图
5      SelectObject (mdc, bg);
6      BitBlt (hdc, 0, 0, 600, 450, mdc, 0, 0, SRCCOPY);
7
8      //贴上处理后的半透明图
9      SelectObject (mdc, girl);
10     BitBlt (hdc, xstart, ystart, 298, 329, mdc, 0, 0, SRCCOPY);
11 }

```

## 程序说明

由于在初始化函数中已经完成了半透明图的处理并重设了前景图 girl 中的内容, 因此, 在

MyPaint()函数被调用时,只进行先贴上背景图再贴上已完成的半透明前景图的操作。

#### 运行结果

程序运行结果如图 2-25 所示。



图 2-25

### 2.2.3 透明半透明效果

上一小节的范例里做出了美观的半透明效果,可是在运行的画面中,似乎可看到前景图四周还留着原来位图的矩形轮廓,感觉有点美中不足。不过没关系,这一小节里将介绍如何制作更完美的透明半透明效果。

制作透明半透明效果还是要运用到前面所讲的透明及半透明技巧,就是先进行透明处理再进行半透明处理。除此之外,还记得前面是怎么做透明的吗?是利用贴图函数直接与已经贴在窗口中的背景图进行两个必要的 Raster 运算完成的。可是,如果这里这样做的话,那结果的透明图已经在窗口上产生了,又要怎么做半透明处理?

在这里多使用了一个内存 DC 与位图对象,先在内存 DC 上完成透明,再取出这个 DC 上的位图内容来进行半透明处理,这样就可以达到目的了。下面直接以一个范例来示范这种做法。这个范例中需要有一张如图 2-26 所示的位图,用来制作前景图的透明。

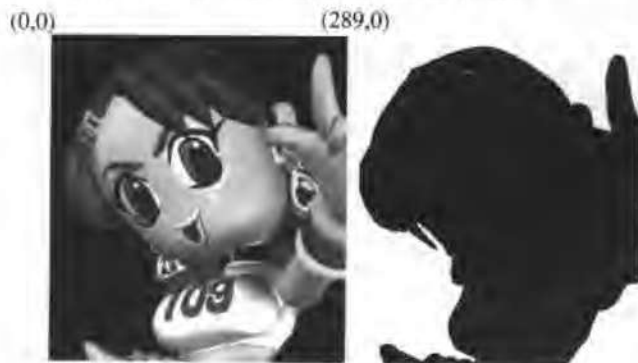


图 2-26

下面以实际范例的说明来介绍如何产生透明半透明的效果。

## » 范例 ch2\_8

先在内存 DC 上完成图案透明,再取出其内容位图进行半透明处理,最后显示透明半透明效果。

程序代码: InitInstance()

```

1  //****初始化函数*****
2  // 透明半透明处理,处理后将半透明图存到 girl 中
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {
5      HWND hWnd;
6      HDC hdc,bufdc;
7      HBITMAP bmp;
8      BITMAP bm1,bm2;
9
10     hInst = hInstance;
11
12     hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
13         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
14
15     if (!hWnd)
16     {
17         return FALSE;
18     }
19
20     MoveWindow(hWnd,10,10,600,450,true);
21     ShowWindow(hWnd, nCmdShow);
22     UpdateWindow(hWnd);
23
24     bg = (HBITMAP)LoadImage(NULL,"bg.bmp",IMAGE_BITMAP,600,450,LR_LOADFROMFILE);
25     bmp = (HBITMAP)LoadImage(NULL,"girlmask.bmp",IMAGE_BITMAP,596,329,
26         LR_LOADFROMFILE);
27
28     GetObject(bg,sizeof(BITMAP),&bm1); //取得bg图信息
29
30     if(bm1.bmBitsPixel != 32 && bm1.bmBitsPixel != 24)
31     {
32         MessageBox(NULL,"此程序只能在 32 bit 或 24 bit 显示模式中运行","警告",0);
33         return FALSE;
34     }
35
36     hdc = GetDC(hWnd);
37     mdc = CreateCompatibleDC(hdc);
38     bufdc = CreateCompatibleDC(hdc);
39     girl = CreateCompatibleBitmap(hdc,298,329);
40
41     SelectObject(mdc, girl);
42
43     //在mdc上进行透明处理

```

```

43 SelectObject(bufdc,bg);
44 BitBlt(mdc,0,0,298,329,bufdc,xstart,ystart,SRCCOPY);
45 SelectObject(bufdc,bmp);
46 BitBlt(mdc,0,0,298,329,bufdc,298,0,SRCAAND);
47 BitBlt(mdc,0,0,298,329,bufdc,0,0,SRCPAINT);
48
49 unsigned char *px1,*px2;
50
51 //处理背景图
52 px1 = new unsigned char [bm1.bmHeight * bm1.bmWidthBytes];
53 GetBitmapBits(bg,bm1.bmHeight * bm1.bmWidthBytes,px1);
54
55 //处理前景图
56 GetObject(girl,sizeof(BITMAP),&bm2);
57 px2 = new unsigned char [bm2.bmHeight * bm2.bmWidthBytes];
58 GetBitmapBits(girl,bm2.bmHeight * bm2.bmWidthBytes,px2);
59
60 int x,y,xend,yend;
61 int i;
62 int rgb_b;
63 int PxBytes = bm1.bmBitsPixel / 8 ;
64
65 xend = xstart + 298;
66 yend = ystart + 329;
67
68 //处理背景图像像素颜色
69 for(y=ystart;y<yend;y++)
70 {
71     for(x=xstart;x<xend;x++)
72     {
73         rgb_b = y * bm1.bmWidthBytes + x * PxBytes ;
74
75         px1[rgb_b] = px1[rgb_b] * 0.7;
76         px1[rgb_b+1] = px1[rgb_b+1] * 0.7;
77         px1[rgb_b+2] = px1[rgb_b+2] * 0.7;
78     }
79 }
80
81 //处理前景图像像素颜色
82 for(y=0;y<(bm2.bmHeight); y++)
83 {
84     for(x=0;x<bm2.bmWidth; x++)
85     {
86         rgb_b = y * bm2.bmWidthBytes + x * PxBytes ;
87         i = (ystart+y) * bm1.bmWidthBytes + (xstart+x) * PxBytes;
88
89         px2[rgb_b] = px2[rgb_b] *0.3 + px1[i];
90         px2[rgb_b+1] = px2[rgb_b+1] *0.3 + px1[i+1];
91         px2[rgb_b+2] = px2[rgb_b+2] *0.3 + px1[i+2];

```

```

92     }
93 }
94
95 SetBitmapBits(girl, bm2.bmHeight*bm2.bmWidthBytes, px2);
96
97 MyPaint(hdc);
98
99 ReleaseDC(hwnd, hdc);
100 DeleteDC(bufdc);
101 DeleteObject(bmp);
102 delete [] px1;
103 delete [] px2;
104
105 return TRUE;
106 }

```

### 程序说明

这个范例在 `InitInstance()` 初始函数中先完成透明的半透明图,再由 `MyPaint()` 函数进行窗口绘图。

- (1) 第 6、7 行: 声明 “bufdc” DC 对象与 “bmp” 位图对象, 在进行透明时会用到。
- (2) 第 24~33 行: 加载位图, 取出 bg 图的信息, 判断系统目前显示模式。
- (3) 第 36~37 行: 建立两个与窗口 DC 兼容的内存 DC “mdc” 与 “bufdc”, 后面的程序会在 bufdc 中选用前景图或背景图, 并贴到 mdc 中进行透明处理。
- (4) 第 38 行: 建立一个与窗口兼容的空位图 “girl”, 其尺寸大小为 298×329 像素, 这个位图在后面程序处理后便是要贴到显示窗口上的透明半透明图案。建立位图的 `CreateCompatibleBitmap()` 函数说明如下。

```

HBITMAP CreateCompatibleBitmap(HDC 窗口 DC,      //建立与窗口兼容的位图
                               int    位图宽,
                               int    位图高);

```

- (5) 第 40 行: 将空位图 girl 存入 mdc 中, 因为要在 mdc 上进行透明处理, 处理完以后原先空的位图将会变成透明图案。

- (6) 第 43~44 行: 将背景图 bg 存入 bufdc 中, 并将透明区域的背景图贴到 mdc 中。此时 mdc 中 girl 位图的内容会随之改变, 这个过程的图示说明如图 2-27 所示。



图 2-27

- (7) 第 45~47 行: 将包含屏蔽图的前景图 bmp 存入 bufdc 中, 再在 mdc 上进行透明处理。此时 mdc 中 girl 位图内容的改变情况如图 2-28 所示。



图 2-28

经过以上的操作,此时 mdc 中的 girl 位图就是我们所要的前景图透明后的结果了。接下来后面的程序再对 girl 位图进行半透明的处理,由于此时人物周围已经变成实际在窗口显示时的背景图,因此半透明处理后的结果就不会像上一小节范例中出现跟背景不协调的矩形轮廓了。

(8) 第 49~95 行: 进行半透明处理。

(9) 第 97~103 行: 调用 MyPaint() 函数绘图, 然后释放资源及内存。

程序代码: MyPaint()

```

1  /*** 自定义绘图函数 *****/
2  void MyPaint (HDC hdc)
3  {
4  * //贴上背景图
5    SelectObject (mdc,bg);
6    BitBlt (hdc,0,0,600,450,mdc,0,0,SRCCOPY);
7
8    //贴上处理后的透明半透明图
9    SelectObject (mdc,girl);
10   BitBlt (hdc,xstart,ystart,298,329,mdc,0,0,SRCCOPY);
11 }

```

#### 程序说明

由于在初始函数中已经完成了透明及半透明处理,且将最后的图案存储在 girl 中,因此并没有修改 MyPaint() 函数,同样也是进行贴上背景图再贴上已处理过的 girl 图的操作。

#### 运行结果

程序运行的结果如图 2-29 所示。



图 2-29

这一节里介绍了 2D 游戏画面贴图特效的内容，下一节开始将进入游戏画面贴图的另一个主题“地图制作”。

## 2.3 游戏地图制作

游戏地图的画面是游戏中不可缺少的重要环节之一，要产生游戏地图，除了可以直接使用已经绘制好的位图外，对于一些画面不太复杂，且具有重复性质的地图或场景，有一个比较好的解决办法，那就是利用地图拼接的方法，将一小块一小块的小地图组合成较大的地图。

地图拼接的优点在于节省系统资源，因为一张大型的地图会占用比较多的内存空间，且加载速度较慢，如果游戏中使用了为数较多的大型地图，那么势必会降低程序运行时的性能，而且需要相当可观的内存空间。

在这一节里，将介绍有关地图拼接的概念，并学习利用不起眼的小地图堆砌出美妙无比的游戏地图的方法。

### 2.3.1 平面地图贴图

首先从最基本的平面地图贴图开始讲起，这种贴图方法相当直观，即利用一张张四方形的小图块组成同样是四方形的大地图，图 2-30 便是一张由 3 种不同图块组合而成的平面地图。

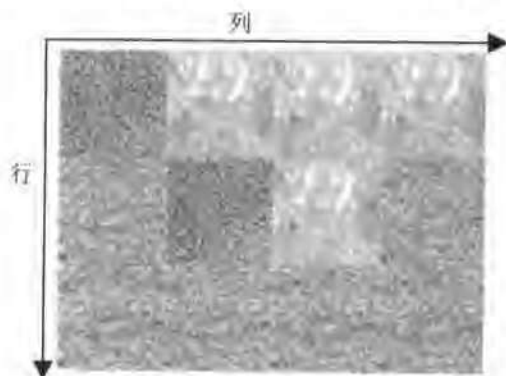


图 2-30

事实上，这张地图是由  $4 \times 3$  张小图块组成的，列方向是 4 张图块，行方向是 3 张图块，这里使用列与行这样的字眼，是因为随后将使用数组来定义地图中出现图块的内容。

从这张图中可以看到，一共出现了 3 种不一样的图块，这是因为程序中会事先以数组来定义哪个位置上要出现哪一种图块，使得拼接出来的地图能够符合需求。现在假设图中 3 种不同图块的编号分别为 0、1 和 2，那么可以以下面的这个一维数组来定义出图 2-30 中的地图。

```
int mapblock[12] = {0,1,1,1,           //第1列
                   2,0,1,2,           //第2列
                   2,2,2,2 };        //第3列
```

将这个一维数组以行列的方式排列，可以看出每个数组元素对应图中的哪个图块。

需要提醒的是，由于使用的是一维数组来定义地图内容，因此上面这个数组的每个元素的索引值是 0……11。但是，由于程序里不论计算图块贴图的位置还是计算整张地图的长宽尺寸，都是以

行列来进行换算的, 所以需要将数组的索引值转换成相应的列编号与行编号, 转换公式如下:

列编号 = 索引值 / 每一列的图块个数 (行数);

行编号 = 索引值 % 每一列的图块个数 (行数);

下面以图 2-31 来验证上面的公式, 方格中的编号是一维数组的元素索引值。

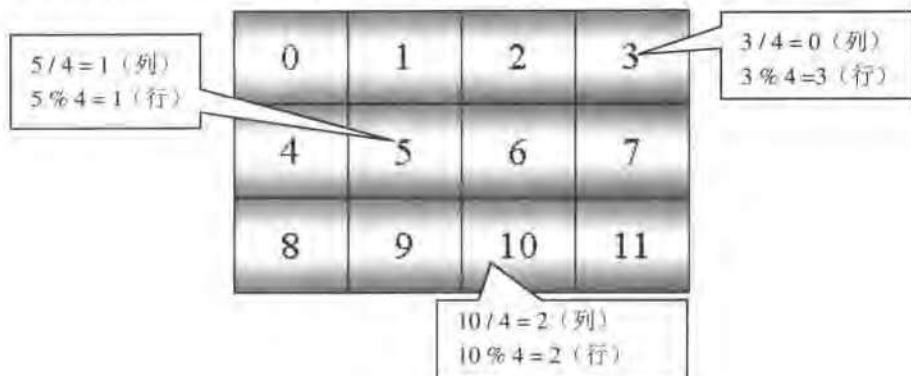


图 2-31

这里要注意的是, 列编号与行编号的起始值是从 0 开始算起, 而一旦算出了列编号与行编号之后, 便可以按照图块的宽与高来求出图块贴图的位置, 下面是计算图块左上点贴图坐标的公式。

左上点 X 坐标 = 行编号 × 图块的宽度;

左上点 Y 坐标 = 列编号 × 图块的高度;

这个公式不难, 就留给读者自行验证。最后来看一个利用图块拼接方法接出一张简单地图的范例。

## » 范例 ch2\_9

运用不同小图块, 示范平面地图拼接的技巧。

**程序代码: 全局变量声明与常数定义**

```
1 //全局变量声明
2 HINSTANCE      hInst;
3 HBITMAP        fullmap;
4 HDC            mdc;
5
6 //常数定义
7 const int rows = 8, cols = 8;
```

**程序说明**

- (1) 第 3 行: 声明 fullmap 位图对象, 在初始函数中完成的地图会存到这个位图中。
- (2) 第 7 行: 定义地图上图块列数 rows 与行数 cols 都是 8。

**程序代码: InitInstance()**

```
1 /***初始化函数***/
2 // 声明地图数组, 进行图块贴图, 完成地图拼接
3 BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4 {
5     HWND hWnd;
```



```

6   HDC hdc,bufdc;
7
8   hInst = hInstance;
9
10  hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
11      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
12
13  if (!hWnd)
14  {
15      return FALSE;
16  }
17
18  MoveWindow(hWnd,10,10,430,450,true);
19  ShowWindow(hWnd, nCmdShow);
20  UpdateWindow(hWnd);
21
22  int mapIndex[rows*cols] = { 2,2,2,2,0,1,0,1, //第1列
23      0,2,2,0,0,0,1,1, //第2列
24      0,0,0,0,0,0,0,1, //第3列
25      2,0,0,0,0,0,2,2, //第4列
26      2,0,0,0,0,2,2,2, //第5列
27      2,0,0,0,2,2,0,0, //第6列
28      0,0,2,2,2,0,0,1, //第7列
29      0,0,2,0,0,0,1,1 }; //第8列
30  hdc = GetDC(hWnd);
31  mdc = CreateCompatibleDC(hdc);
32  bufdc = CreateCompatibleDC(hdc);
33  fullmap = CreateCompatibleBitmap(hdc,cols*50,rows*50);
34
35  SelectObject(mdc,fullmap);
36
37  HBITMAP map[3];
38  char filename[20] = "";
39  int rowNum,colNum;
40  int i,x,y;
41
42  //加载各图块位图
43  for(i=0;i<3;i++)
44  {
45      sprintf(filename,"map%d.bmp",i);
46      map[i] = (HBITMAP)LoadImage(NULL,filename,IMAGE_BITMAP,50,50,
47          LR_LOADFROMFILE);
48  }
49  //按照mapIndex 数组中的定义取出对应图块, 进行地图拼接
50  for (i=0;i<rows*cols;i++)
51  {
52      SelectObject(bufdc,map[mapIndex[i]]);
53

```

```

54     rowNum = i / cols;          //求列编号
55     colNum = i % cols;          //求行编号
56     x = colNum * 50;            //求贴图 X 坐标
57     y = rowNum * 50;            //求贴图 Y 坐标
58
59     BitBlt(mdc,x,y,50,50,bufdc,0,0,SRCCOPY);
60 }
61
62 MyPaint(hdc);
63
64 ReleaseDC(hWnd,hdc);
65 DeleteDC(bufdc);
66
67 return TRUE;
68 }

```

#### 程序说明

在初始函数中，先在内存 DC(mdc)上完成地图拼接，完成后的地图为 fullmap。

(1) 第 22~29 行：定义地图上各个图块的内容。

(2) 第 33 行：先建立“fullmap”为一空白的位图，其宽与高分别为“行数×图块宽”与“列数×图块高”。

(3) 第 35 行：将 fullmap 存入 mdc 中。

(4) 第 43~47 行：利用循环转换图文件名，取出各个图块存于“map[i]”中。图块文件名为“map0.bmp”和“map1.bmp”等。

(5) 第 52 行：根据 mapIndex[i]中的代号选取对应的图块到 bufdc 中。代号为“0”则取“map[0]”，代号为“1”则取“map[1]”等，依次类推。

(6) 第 54~59 行：以公式求出行编号与列编号，并计算图块的贴图坐标，在 mdc 上进行贴图。

(7) 第 62 行：当第 50~60 行的循环完成在 mdc 上的图块贴图之后，fullmap 便是拼接出来的地图，此时再调用 MyPaint()函数进行窗口贴图。

#### 程序代码：MyPaint()

```

1  //****自定义绘图函数*****
2  void MyPaint(HDC hdc)
3  {
4      //贴上拼接后的组合地图
5      SelectObject(mdc,fullmap);
6      BitBlt(hdc,10,10,cols*50,rows*50,mdc,0,0,SRCCOPY);
7  }

```

#### 程序说明

第 5~6 行：在窗口中贴上拼接后的组合地图，整个地图的贴图大小按照拼接地图的行数、列数和图块的宽与高来决定。

#### 运行结果

程序运行结果如图 2-32 所示。



图 2-32

这个范例程序具有一定的灵活性，只要更改常数中列数与行数的值，并重新定义 `mapIndex[]` 数组中的值，便可以组合出大小尺寸及内容不尽相同的平面地图来。

## 2.3.2 斜角地图贴图

斜角地图其实是平面地图的一种变化，它是将拼接地图的图块内容，由原先的四方形图案改变成由  $45^\circ$  角俯瞰四方形图案时的菱形图案，由这些菱形图案所拼接完成后的地图，就是一张由  $45^\circ$  角俯瞰的斜角地图了。

斜角地图拼接方法同样也是使用与平面地图一样的行与列的方法，事实上它们的原理是一样的。但由于地图拼接时只要取用位图中的菱形部分，因此在贴图坐标的计算上会有所不同。下面就来说明菱形图块与方形图块在贴图时的差异，如图 2-33 所示，其中的数字是图块的编号。



图 2-33

图 2-33 中的左边的图是四方形图块的拼接，而右边的图则是菱形图块的拼接。四方形图块拼接方法是：图块编号换算成行编号与列编号再换算成贴图坐标。对于斜角地图拼接来说，这些步骤都是一样的，但是在换算贴图坐标时，由于只要显示图块中的菱形部分，因此在贴图排列的方式上会有不同，因而贴图坐标的计算公式也就不一样了。

此外，在合并两个图块的菱形部分时，还需要加上一步透明的步骤，不然若直接按照求得的贴图坐标来进行贴图，其效果就会像如图 2-34 所示。



图 2-34

接下来看看斜角地图拼接时,各个图块编号与实际排列的情形,如图2-35所示。

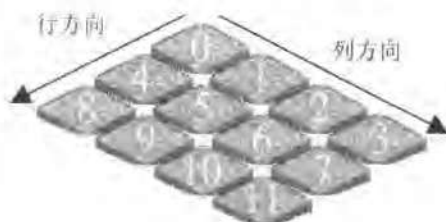


图 2-35

图2-35同样是一张由 $4 \times 3$ 个小图块所拼接而成的地图,其中的数字是图块编号。对于每一图块首先必须算出它的行编号与列编号,然后才能计算它实际的贴图坐标,计算行列编号的方法与上一小节所使用的公式一样,即:

列编号 = 索引值 / 每一列的图块个数 (行数);

行编号 = 索引值 % 每一列的图块个数 (行数);

求出了行编号与列编号后,就可以计算出图块贴图时左上点的坐标,除此之外,还需要知道图块中菱形部分的长度与高度,这里假设图块中菱形的宽与高分别是 $w$ 与 $h$ ,如图2-36所示。

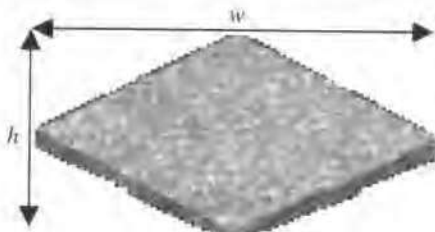


图 2-36

那么图块左上点贴图坐标的计算公式如下。

左上点 X 坐标 =  $xstart + 行编号 \times w/2 - 列编号 \times (w/2)$ ;

左上点 Y 坐标 =  $ystart + 列编号 \times h/2 + 行编号 \times h/2$ ;

公式中的  $xstart$  与  $ystart$  是代表第一张图块左上角贴图坐标的位置,以图2-37来说明这个公式。

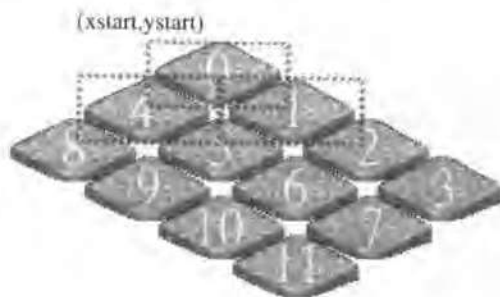


图 2-37

图中以虚线框来表示图块真正的矩形范围,在进行贴图时,首先要自定义第1张图块的贴图位置,其他图块的贴图坐标再由此图块向下延伸。现在假设给定图块0的贴图坐标是 $(xstart, ystart)$ ,那么接下来考虑图块1的矩形范围,它左上角贴图的坐标则是 $(xstart + w/2, ystart + h/2)$ ,考虑图块2

的矩形范围,它左上角贴图的坐标又变成 $(xstart+w/2 \times 2, ystart+h/2 \times 2)$ 。依次类推,再加入行编号与列编号,可以得到下面的这个求图块贴图坐标的公式:

左上点 X 坐标 =  $xstart + \text{行编号} \times w/2$ ;

左上点 Y 坐标 =  $ystart + \text{列编号} \times h/2$ ;

但是要注意一点,这是当图块都在属于同一列的情况。考虑下一列的图块 4,图块 4 的左上角贴图坐标是 $(xstart-w/2, ystart+h/2)$ ,而图块 5 的左上角贴图坐标则是 $(xstart-w/2+w/2, ystart+h/2+h/2)$ ,图块 6 的左上角贴图坐标为 $(xstart-w/2+w/2 \times 2, ystart+h/2+h/2 \times 2)$ ,依次类推,可看出同一列上坐标变化规律都是一样的,贴图坐标都是往右下方递增半个图块的长与高单位。

如果是在同一行(图块 0、4、8)上的坐标变化则是往左下方递减半个图块的长(X 轴方向)以及递增半个图块的高(Y 轴方向),因此利用图块的行编号与列编号便得出了前面的贴图坐标公式。

计算出每个图块的坐标并完成了斜角地图的拼接后,此时要将整块地图贴到窗口中,还需要知道地图的宽度与高度,计算的方法可通过图 2-38 进行说明。

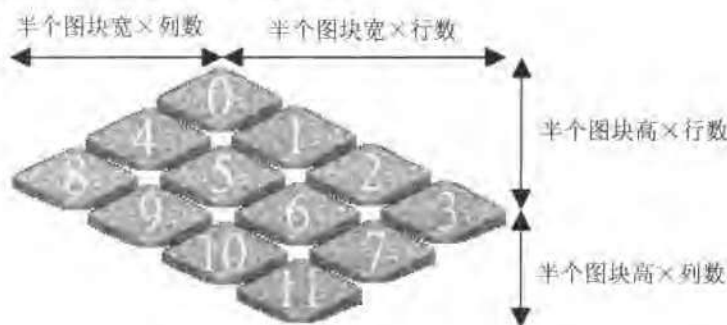


图 2-38

由上面可以很容易的推导出整张地图的宽与高计算公式如下:

地图宽 =  $(\text{列数} + \text{行数}) \times w/2$ ;

地图高 =  $(\text{列数} + \text{行数}) \times h/2$ ;

在了解了关于斜角地图拼接的方法之后,接下来的这个范例将上一小节里的平面拼接地图转换成以  $45^\circ$  角俯视的斜角地图。

## » 范例 ch2\_10

运用菱形图案的图块,示范斜角地图拼接的制作方法。

**程序代码: 全局变量声明与常数定义**

```
1 //全局变量声明
2 HINSTANCE hInst;
3 HBITMAP fullmap;
4 HDC hdc;
5
6 //常数定义
7 const int rows = 10, cols = 10;
```

**程序说明**

- (1) 第 3 行: 声明 fullmap 位图对象,在初始函数中完成的斜角地图会保存到这个位图中。
- (2) 第 7 行: 定义地图上图块的列数 rows 与行数 cols 都为 10。

## 程序代码: InitInstance()

```

1  //****初始化函数*****
2  // 声明地图数组, 进行贴图, 完成地图拼接
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {
5      HWND hWnd;
6      HDC hdc,bufdc;
7
8      gInst = hInstance;
9
10     hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
11         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
12
13     if (!hWnd)
14     {
15         return FALSE;
16     }
17
18     MoveWindow(hWnd,10,10,640,480,true);
19     ShowWindow(hWnd, nCmdShow);
20     UpdateWindow(hWnd);
21
22     int mapIndex[rows*cols] = { 2,2,2,2,2,0,1,0,1,0, //第1列
23         3,3,2,2,0,0,0,1,1,0, //第2列
24         3,0,0,0,0,0,0,0,1,2, //第3列
25         2,2,0,0,0,0,0,2,2,2, //第4列
26         2,2,0,0,0,0,2,2,2,2, //第5列
27         2,2,0,0,0,2,2,0,0,2, //第6列
28         2,0,0,2,2,2,0,0,1,0, //第7列
29         0,0,2,0,0,0,1,1,1,1, //第8列
30         0,2,0,3,3,3,3,3,3,1, //第9列
31         2,0,3,3,3,3,3,3,3,3 }; //第10列
32
33     hdc = GetDC(hWnd);
34     mdc = CreateCompatibleDC(hdc);
35     bufdc = CreateCompatibleDC(hdc);
36
37     HBITMAP map[4];
38     char filename[20] = "";
39     int rowNum,colNum;
40     int i,x,y;
41     int xstart,ystart;
42
43     xstart = 32 * (rows-1);
44     ystart = 0;
45
46     fullmap = (HBITMAP)LoadImage(NULL, "bg.bmp", IMAGE_BITMAP, 640, 480,
        LR_LOADFROMFILE);

```

```

47 SelectObject(mdc, Fullmap);
48
49 //加载各图块位图
50 for(i=0;i<4;i++)
51 {
52     sprintf(filename, "map%d.bmp", i);
53     map[i] = (HBITMAP)LoadImage(NULL, filename, IMAGE_BITMAP, 128, 32,
        LR_LOADFROMFILE);
54 }
55
56 //按照mapIndex 数组中的定义取出对应图块, 进行地图拼接
57 for (i=0;i<rows*cols;i++)
58 {
59     SelectObject(bufdc, map[mapIndex[i]]);
60
61     rowNum = i / cols;    //求列编号
62     colNum = i % cols;    //求行编号
63     x = xstart + colNum * 32 + rowNum * (-32); //求贴图 X 坐标
64     y = ystart + rowNum * 16 + colNum * 16;    //求贴图 Y 坐标
65
66     BitBlt(mdc, x, y, 64, 32, bufdc, 64, 0, SRCAND);
67     BitBlt(mdc, x, y, 64, 32, bufdc, 0, 0, SRCPAINT);
68 }
69
70 MyPaint(mdc);
71
72 ReleaseDC(hWnd, hdc);
73 DeleteDC(bufdc);
74
75 return TRUE;
76 }

```

### 程序说明

在初始化函数中, 先在内存 DC(mdc)上完成地图拼接, 完成后的地图则是 fullmap。

(1) 第 22~31 行: 定义 10×10 地图上各个图块的内容。

(2) 第 43~44 行: 设定第 1 个图块的初始坐标, 设定 X 轴上的坐标 xstart 是将图块放在整个拼接地图的中间位置。

(3) 第 46~47 行: 加载背景图, 并选用至 mdc 中。

(4) 第 50~54 行: 取出各个图块并存于“map[i]”中。

(5) 第 59 行: 根据“mapIndex[i]”中的代号选取对应的图块到 bufdc 中。

(6) 第 61~64 行: 求出行编号与列编号, 并计算图块的贴图坐标, 在 mdc 上进行透明贴图。这里使用的是宽 64 和高 32 的图块, 如图 2-39 所示。

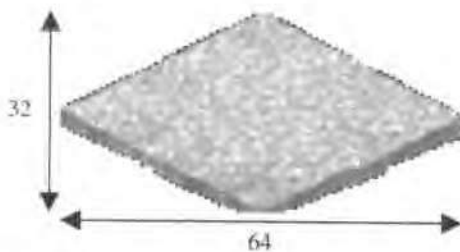


图 2-39

在循环运行完成后，fullmap 便是已经完成拼接的斜角地图。

**程序说明：**MyPaint()

```
1  //****自定义绘图函数*****  
2  void MyPaint(HDC hdc)  
3  {  
4      //贴上拼接后的组合地图  
5      SelectObject(hdc, fullmap);  
6      BitBlt(hdc, 0, 0, 640, 480, hdc, 0, 0, SRCCOPY);  
7  }
```

**程序说明**

调用 MyPaint()函数时会将 fullmap 贴到窗口中，显示已完成的拼接地图。

**运行结果**

程序运行结果如图 2-40 所示。

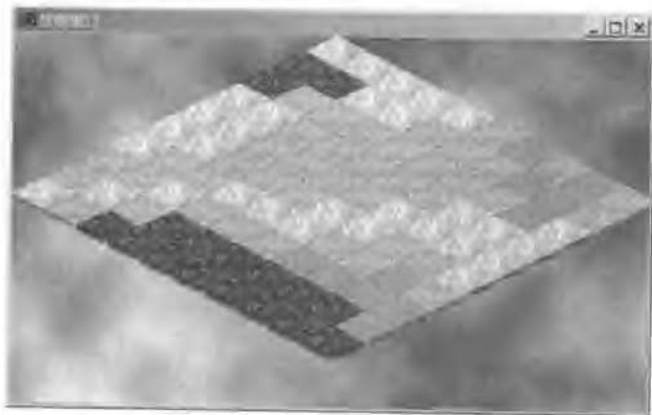


图 2-40

这样，变成  $45^\circ$  角俯视的斜角地图看起来很有立体感。

### 2.3.3 景物贴图

学会了游戏地图的拼接技巧，这一小节的主题将要介绍如何在地图上布置一些景物，如花草树木和房子等。景物的点缀将使游戏地图更美观。

其实一旦完成了地图的拼接，景物部分就容易多了。同样可使用一个与地图数组相同大小的数组来定义哪个图块位置上要出现哪些景物，但由于景物图的大小与图块的大小并不一定相同，因此



还要再将景物贴图的坐标稍做修正，使得这些景物可以出现在正确的位置上。下面以在  $64 \times 32$  的斜角图块上贴上一张  $50 \times 60$  的树木图来做说明，如图 2-41 所示。

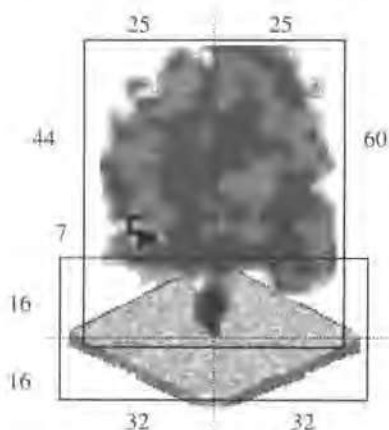


图 2-41

从图中可以看出，若斜角图块的贴图坐标是  $(x, y)$ ，那么树木图的  $X$  坐标必须向右移动  $32-25=7$  个单位， $Y$  坐标则必须向上移动  $60-16=44$  个单位，则树木图的贴图坐标为  $(x+7, y-44)$ 。按照这样的方法，再对其他景物实际的贴图坐标进行修正，最后就可以得到所要的地图场景了。

接下来的范例，会在上一个范例里的斜角地图中加入两个不同的景物，展现不同的新面貌。

## 》范例 ch2\_11

在斜角地图中加入景物，展现游戏地图效果。

程序代码：InitInstance()

```

1  //****初始函数*****
2  // 声明地图及景物数组，完成地图拼接及景物布置
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {
5      HWND hWnd;
6      HDC hdc, bufdc;
7
8      hInst = hInstance;
9
10     hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
11         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
12
13     if (!hWnd)
14     {
15         return FALSE;
16     }
17
18     MoveWindow(hWnd, 10, 10, 640, 400, true);
19     ShowWindow(hWnd, nCmdShow);
20     UpdateWindow(hWnd);
    
```

```

21
22 int mapIndex[rows*cols]={2,2,2,2,2,0,1,0,1,0,      //第1列
23                          3,3,2,2,0,0,0,1,1,0,      //第2列
24                          3,0,0,0,0,0,0,0,1,2,      //第3列
25                          2,2,0,0,0,0,0,2,2,2,      //第4列
26                          2,2,0,0,0,0,2,2,2,2,      //第5列
27                          2,2,0,0,0,2,2,0,0,2,      //第6列
28                          2,0,0,2,2,2,0,0,1,0,      //第7列
29                          0,0,2,0,0,0,1,1,1,1,      //第8列
30                          0,2,0,3,3,3,3,3,3,1,      //第9列
31                          2,0,3,3,3,3,3,3,3,3 };      //第10列
32
33 int sceneIndex[rows*cols] = { 0,2,2,0,2,0,1,0,1,1,    //第1列
34                               0,0,0,0,0,0,0,1,1,0,    //第2列
35                               0,0,0,0,0,0,1,0,1,0,    //第3列
36                               0,0,1,0,1,0,0,0,2,0,    //第4列
37                               2,2,0,0,1,0,0,0,0,2,    //第5列
38                               0,0,0,0,0,0,0,0,0,0,    //第6列
39                               0,0,1,0,0,0,0,0,1,0,    //第7列
40                               0,0,0,0,0,0,1,1,1,1,    //第8列
41                               1,0,0,0,0,0,0,0,0,1,    //第9列
42                               2,0,0,0,0,0,0,0,0,0 };    //第10列
43
44 hdc = GetDC(hWnd);
45 mdc = CreateCompatibleDC(hdc);
46 bufdc = CreateCompatibleDC(hdc);
47
48 HBITMAP map[4],scene[2];
49 char filename[20] = "";
50 int rowNum,colNum;
51 int i,x,y;
52 int xstart,ystart;
53
54 xstart = 32 * (rows-1);
55 ystart = 0;
56
57 fullmap = (HBITMAP)LoadImage(NULL,"bg.bmp",IMAGE_BITMAP,640,480,
58                               LR_LOADFROMFILE);
59 SelectObject(mdc,fullmap);
60
61 //加载各图块位图
62 for(i=0;i<4;i++)
63 {
64     sprintf(filename,"map%d.bmp",i);
65     map[i] = (HBITMAP)LoadImage(NULL,filename,IMAGE_BITMAP,128,32,
66                                 LR_LOADFROMFILE);
67 }

```

```

68 //加载各场景图
69 for(i=0;i<2;i++)
70 {
71     sprintf(filename,"scene%d.bmp",i+1);
72     scene[i] = (HBITMAP)LoadImage(NULL,filename,IMAGE_BITMAP,100,60,
        LR_LOADFROMFILE);
73 }
74
75 //按照mapIndex 数组中的定义取出对应图块,进行地图拼接
76 for (i=0;i<rows*cols;i++)
77 {
78     SelectObject(bufdc,map[mapIndex[i]]);
79
80     rowNum = i / cols;    //求列编号
81     colNum = i % cols;    //求行编号
82     x = xstart + colNum * 32 + rowNum * (-32);    //求贴图 X 坐标
83     y = ystart + rowNum * 16 + colNum * 16;    //求贴图 Y 坐标
84
85     BitBlt(mdc,x,y,64,32,bufdc,64,0,SRCAAND);
86     BitBlt(mdc,x,y,64,32,bufdc,0,0,SRCPAINT);
87
88     switch(sceneIndex[i])
89     {
90         case 1:    //树木贴图
91             SelectObject(bufdc,scene[0]);
92             BitBlt(mdc,x+7,y-44,50,60,bufdc,50,0,SRCAAND);
93             BitBlt(mdc,x+7,y-44,50,60,bufdc,0,0,SRCPAINT);
94             break;
95         case 2:    //房子贴图
96             SelectObject(bufdc,scene[1]);
97             BitBlt(mdc,x+7,y-30,50,60,bufdc,50,0,SRCAAND);
98             BitBlt(mdc,x+7,y-30,50,60,bufdc,0,0,SRCPAINT);
99             break;
100     }
101 }
102
103 MyPaint(hdc);
104
105 ReleaseDC(hWnd,hdc);
106 DeleteDC(bufdc);
107
108 return TRUE;
109 }

```

### 程序说明

(1) 第 22~31 行: 定义地图数组。

(2) 第 33~42 行: 定义场景数组, 其中“0”表示图块上不出现景物,“1”表示图块上出现树木,“2”表示图块上出现房子。

(3) 第 69~73 行: 加载景物图案。

(4) 第 76~101 行: 在进行地图拼接处理每一个图块时, 判断该图块上是否出现景物, 并进行必要的景物贴图。

(5) 第 88~100 行: 判断出现的景物类型, 进行透明贴图, 贴图坐标按照图块的贴图坐标  $x$ 、 $y$  进行修正。

### 运行结果

程序运行结果如图 2-42 所示。



图 2-42

这一章中介绍了 2D 游戏所用到的画面特效处理方法及拼接地图的设计方法, 奠定了游戏设计的基础。下一章将更进一步讨论动画产生的方法, 让静态的图案动起来。

## 课后重点整理

- GDI (Graphics Device Interface) 中文可译为“图形设备接口”, 是 Windows API 的成员, 管理所有显像设备的图像显示及输出功能。
- 对一个游戏程序来说, 窗口建立之后, 显示的屏幕上便划分为 3 个区域, 即屏幕区 (Screen)、窗口区 (Window) 与内部窗口区 (Client)。
- Device Context (设备内容) 一般常称为 DC, 就绘图的观点来说, DC 就是程序可以进行绘图的地方。举例来说, 如果要在整个屏幕区上绘图, 那么 Device (设备) 就是屏幕, 而 Device Context 就是屏幕区上的绘图层。
- 画笔与画刷都是 GDI 中所定义的图形对象, 画笔是线条的样式, 画刷则是封闭图形内部的填充样式。
- GDI 对象一经建立便会占用部分内存, 一旦不使用的时候务必将它们删除。
- GDI 函数中关于多边形的绘图函数有以下几个: Polygon、PolyLine、PolylineTo、PolyPolygon 和 PolyPolyline。
- 扇形与弓形都有连接的起点与终点, 不同之处在于, 扇形还会与椭圆的中心点相连接, 而弓形则是直接连接起点与终点。绘制扇形的函数为 Pie(), 绘制弓形的函数则为 Chord()。
- 从文件加载位图到绘制窗口中必须经过以下几个步骤:
  - (1) 从文件加载位图 (BITMAP) 对象;

- (2) 建立一个与窗口 DC 兼容的内存 DC;
- (3) 内存 DC 使用步骤 (1) 所建立的位图对象;
- (4) 将内存 DC 的内容贴到窗口 DC 中便完成图像的显示。

- 程序产生半透明效果的实际操作步骤如下。
  - 步骤一：取得位图结构
  - 步骤二：建立暂存数组
  - 步骤三：取得位图位值
  - 步骤四：合成像素颜色值
  - 步骤五：重设位图颜色
- 要产生游戏地图，除了可以直接使用已经绘制好的位图外，对于一些画面不很复杂，且具有重复性质的地图或场景，有一个比较好的解决办法，那就是利用地图拼接的方法，将一小块一小块的小地图组合成更大型的地图。
- 地图拼接的优点在于节省系统资源，因为一张大型的地图会占用比较多的内存空间，且加载速度较慢。
- 斜角地图是将拼接地图的图块内容，由原先的四方形图案改变成好像是由 45° 角俯瞰四方形图案时的菱形图案，再由这些菱形图案拼接完成后的地图。

## 课后练习

1. 试以图例说明在窗口程序中，显示的屏幕上可划分为 3 个区域：屏幕区 (Screen)、窗口区 (Window) 与内部窗口区 (Client)，试区别它们之间的范围差异。
2. 请简述一种透明图的制作方法。
3. 试简述半透明的制作原理。
4. BitBlt() 函数最后一个参数所输入的是称为 Raster 的运算值，这个值是用来设定内存 DC 到目的地 DC 的贴图方式的。表 2-4 中列出了几个可使用的 Raster 值，请说明其意义。

表 2-4

Raster 值	说 明
BLACKNESS	
DS INVERT	
MERGE COPY	
SRCCOPY	
SRCErase	
SRC INVERT	
SRC PAINT	

## 第 3 章 游戏动画技巧

### 3.1 基础动画显示

游戏中播放动画的方法有两种：一种是直接播放影片文件（如 AVI 和 MPEG）文件，常用在游戏的片头与片尾；另一种则是游戏进行时利用连续贴图的方式，制作动画的效果。事实上游戏程序本身几乎都是以无限循环的方式不断地在游戏窗口中进行窗口画面重绘的操作，即使画面没有任何变化，这个重绘的操作还是会不断地进行，一直到玩家选择结束游戏为止。

这一节将介绍如何利用 Windows 本身的定时器及游戏中常用的游戏循环来制作游戏的动态效果，并介绍最为常见的透明动画的制作方法。

#### 3.1.1 定时器的使用

定时器（Timer）对象可以每隔一段时间发出一个时间消息，程序一旦接收到此消息之后，便可以决定接下来要做哪些事情。这样的一个特性刚好可以适合播放静态的连续图片，产生动画的效果。下面来介绍如何建立与使用定时器。

##### 1. 建立定时器

Windows API 的 SetTimer() 函数可为窗口建立一个定时器，并且每隔一段时间就发出 WM\_TIMER 消息，此函数的使用语法如下。

UINT SetTimer(	HWND	接收定时器消息的窗口，
	UINT	定时器代号，
	UINT	时间间隔，
	TIMERPROC	处理响应函数)；

SetTimer() 函数的第 2 个参数是定时器的代号，这个代号在同一个窗口中必须是惟一的，且值不为 0；第 3 个参数则是定时器发出 WM\_TIMER 消息的时间间隔，以千分之一秒为单位，也就是若此参数设为 1000，则每间隔 1 秒发出一个 WM\_TIMER 消息；第 4 个参数则用于设定由系统调用处理 WM\_TIMER 消息的响应函数，如果不用响应函数处理 WM\_TIMER 消息，则此参数应设为 NULL。

下面是设定一个每隔 0.5 秒发出 WM\_TIMER 消息的定时器的程序代码。

```
SetTimer(1, 500, NULL);
```

如果不使用响应函数来处理定时器的消息，那么就必须在消息处理函数中定义处理消息的程序代码。

## 2. 删除定时器

定时器建立后, 就会一直自动地按照设定的时间间隔发出 WM\_TIMER 消息, 如果要停用某个定时器, 必须使用下面的这个函数。

```
BOOL KillTimer( int 定时器代号 );
```

在大致了解了定时器的使用方法后, 接下来将运用定时器使预先准备的几张人物连续摆动的位图, 产生动画的效果, 如图 3-1 所示。



图 3-1

## 》》 范例 ch3\_1

使用定时器, 将连续的人物图案显示在窗口上, 产生动画效果。

### 程序代码: 全局变量声明

```
1  //全局变量声明
2  HINSTANCE  hInst;
3  HBITMAP  girl[7];
4  HDC      hdc, hdc;
5  int      num;
```

### 程序说明

- (1) 第 3 行: 声明位图数组用来存储各张人物位图。
- (2) 第 4 行: 声明 “hdc” 为全局变量, 用来存储窗口 DC, 这样后面的程序进行动画绘图会较为方便。
- (3) 第 5 行: “num” 变量用来记录目前显示的图号。

### 程序代码: InitInstance()

```
6  //****初始化函数****
7  // 1.从文件加载位图
8  // 2.建立定时器
9  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
10 {
11  HWND hWnd;
12  char filename[20] = "";
13  int i;
14
15  hInst = hInstance;
```

```

16
17 hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
18     (W_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
19
20 if (!hWnd)
21 {
22     return FALSE;
23 }
24
25 MoveWindow(hWnd, 10, 10, 600, 450, true);
26 ShowWindow(hWnd, nCmdShow);
27 UpdateWindow(hWnd);
28
29 hdc = GetDC(hWnd);
30 mdc = CreateCompatibleDC(hdc);
31
32 //载入各个人物位图
33 for(i=0;i<7;i++)
34 {
35     sprintf(filename, "girl%d.bmp", i);
36     girl[i] = (HBITMAP)LoadImage(NULL, filename, IMAGE_BITMAP, 640, 480,
37         LR_LOADFROMFILE);
38 }
39 num = 0;
40 SetTimer(hWnd, 1, 500, NULL);
41
42 MyPaint(hdc);
43
44 return TRUE;
45 }

```

#### 程序说明

- (1) 第28~32行：载入各个人物位图。
- (2) 第34行：设定初始的显示图号为“0”。
- (3) 第35行：建立定时器，间隔0.5秒发出消息。

#### 程序代码：WndProc

```

1  //****消息处理函数*****
2  // 1. 加入处理WM_TIMER消息
3  // 2. 在窗口结束时删除定时器
4  LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
5      lParam)
6  {
7      int i;
8      switch (message)
9      {
10         case WM_TIMER:
11             //时间消息

```



```

11         MyPaint(hdc);
12         break;
13     case WM_DESTROY:                                //窗口结束消息
14         DeleteDC(mdc);
15         ReleaseDC(hWnd,hdc);
16         for(i=0;i<7;i++)
17             DeleteObject(girl[i]);
18
19         KillTimer(hWnd,1);
20
21         PostQuitMessage(0);
22         break;
23     default:                                          //其他消息
24         return DefWindowProc(hWnd, message, wParam, lParam);
25 }
26 return 0;
27 }

```

## 程序说明

在消息处理函数中，由于已经设定了一个定时器让程序每隔一定间隔便进行窗口的重绘操作，因此在这里不再需要特别去处理 WM\_PAINT 消息。

(1) 第 10~12 行：在消息循环中加入处理 WM\_TIMER 消息，当接收到此消息时便调用 MyPaint()函数进行窗口绘图。

(2) 第 19 行：在窗口结束时，删除所建立的定时器。

## 程序代码：MyPaint()

```

1  //****自定义绘图函数*****
2  // 按照图号顺序进行窗口贴图
3  void MyPaint(HDC hdc)
4  {
5      if(num > 6)
6          num = 0;
7      SelectObject(mdc, girl[num]);
8      BitBlt(hdc, 0, 0, 600, 450, mdc, 0, 0, SRCCOPY);
9
10     num++;
11 }

```

## 程序说明

(1) 第 5~6 行：判断目前图号是否已超过最大图号，若超过最大图号“6”，则将显示图号重设为“0”。

(2) 第 7~8 行：依目前图号进行窗口贴图。

(3) 第 10 行：将“num”值加 1，为下一次要显示的图号。

## 运行结果

程序运行结果如图 3-2 所示。



图 3-2 显示人物动画

### 3.1.2 游戏循环

定时器的使用固然很简单方便，但是事实上这样的方法仅适合用在显示简易动画及小型的游戏程序中。因为一般而言，游戏本身需要显示顺畅的游戏画面，使玩家感觉不到延迟的状况。基本上游戏画面必须在一秒钟之内更新至少 25 次以上，这一秒钟内程序还必须进行消息的处理和大量数学运算甚至音效的输出等操作。而使用定时器的消息来驱动这些操作，往往达不到所要求的标准，不然就会产生画面显示不顺畅和游戏响应时间太长的情况。

这里提出一种“游戏循环”概念。游戏循环是将原先程序中的消息循环加以修改，方法是判断其中的内容目前是否有要处理的消息，如果有则进行处理，否则按照设定的时间间隔来重绘画面。下面是接下来内容里所使用的游戏循环的程序代码。

```
1      //游戏循环
2      while( msg.message!=WM_QUIT )
3      {
4          if( PeekMessage( &msg, NULL, 0,0 ,PM_REMOVE) )    //检测消息
5          {
6              TranslateMessage( &msg );
7              DispatchMessage( &msg );
8          }
9          else
10         {
11             tNow = GetTickCount();                          //取得目前时间
12             if(tNow-tPre >= 40)
13                 MyPaint(hdc);
14         }
15     }
```

下面来说明游戏循环程序片段中的几个重点。

(1) 第 2 行：当收到的 msg.message 不是窗口结束消息 WM\_QUIT，则继续运行循环。其中 msg 是一个 MSG 的消息结构，其结构成员 message 则是一个消息类型的代号。

(2) 第 4 行：使用 PeekMessage() 函数来检测目前是否有要处理的消息，若检测到消息（包含 WM\_QUIT 消息）则会返回一个非“0”的值，否则返回“0”。因此在游戏循环中，若检测到消息便进行消息的处理，否则运行 else 叙述之后的程序代码。

这里要注意的是，PeekMessage() 函数不能用原先消息循环的条件 GetMessage() 取代，因为

GetMessage()函数只有在取得 WM\_QUIT 消息时才会返回“0”，其他时候则是返回非“0”值或“-1”（发生错误时）

(3) 第 11 行：GetTickCount()函数会取得系统开始运行到目前所经过的时间，单位是百万分之一秒。

```
DWORD GetTickCount(); //取得系统开始到目前经过的时间
```

在这里取得时间的目的主要是可以搭配接下来的判断式，用来调整游戏运行的速度，使得游戏不会因为运行计算机速度的不同而跑得太快或太慢。

(4) 第 11~12 行：在第 12 行的条件式中，“tPre”记录前次绘图的时间，而“tNow-tPre”则是计算上次绘图到这次循环运行之间相差多少时间。这里设置为若相差 40 个单位时间以上则再次进行绘图的操作，通过这个数值的控制可以调整游戏运行的快慢。

这里设定 40 个单位时间（1 微秒）的原因是，因为每隔 40 个单位进行一次绘图的操作，那么 1 秒钟大约重绘窗口  $1000/40=25$  次，刚好可以达到期望值。

由于循环的运行速度远比定时器发出时间信号来得快，因此使用游戏循环可以更精准地控制程序运行速度并提高每秒钟画面重绘的次数。

了解了游戏循环使用的基本概念之后，接下来的范例将以游戏循环的方法进行窗口的连续贴图，更精确地制作游戏动画效果。

## 》范例 ch3\_2

使用游戏循环产生动画效果，并在窗口左上角显示每秒画面更新次数。

### 程序代码：全局变量声明

```
1 //全局变量声明
2 HINSTANCE hInst;
3 HBITMAP man[7];
4 HDC hdc,mdc;
5 HWND hWnd;
6 DWORD tPre,tNow,tCheck;
7 int num,frame,fps;
```

### 程序说明

(1) 第 6 行：声明 3 个变量，用来记录时间，说明如表 3-1 所示。

表 3-1

变量名称	用途
tPre	记录上一次绘图的时间
tNow	记录此次准备绘图的时间
tCheck	记录每秒开始的时间

(2) 第 7 行：“num”变量用来记录图号；“frame”变量用来累加每次画面更新的次数；“fps（frame per second）”变量用来记录每秒画面更新的次数。

### 程序代码：WinMain()

```
1 //*****主程序*****
2 int APIENTRY WinMain(HINSTANCE hInstance,
```

```

3             HINSTANCE hPrevInstance,
4             LPSTR      lpCmdLine,
5             int         nCmdShow)
6 {
7     MSG msg;
8
9     MyRegisterClass(hInstance);
10
11    //运行初始化函数
12    if (!InitInstance (hInstance, nCmdShow))
13    {
14        return FALSE;
15    }
16
17    //游戏循环
18    while( msg.message!=WM_QUIT )
19    {
20        if( PeekMessage( &msg, NULL, 0,0 ,PM_REMOVE) )
21        {
22            TranslateMessage( &msg );
23            DispatchMessage( &msg );
24        }
25        else
26        {
27            tNow = GetTickCount();
28            if(tNow-tPre >= 100)
29                MyPaint(hdc);
30        }
31    }
32
33    return msg.wParam;
34 }

```

#### 程序说明

- (1) 第 17~31 行：在主程序中加入游戏循环。
- (2) 第 27~28 行：当此次循环运行与上次绘图时间相差 0.1 秒时再进行重绘操作。

#### 程序代码：MyPaint()

```

1    //****自定义绘图函数*****
2    // 1.计算与显示每秒画面更新次数
3    // 2.按照图号顺序进行窗口贴图
4    void MyPaint(HDC hdc)
5    {
6        char str[40] = "";
7
8        if(num == 7)
9            num = 0;
10       frame++;           //画面更新次数加1
11       if(tNow - tCheck >= 1000)

```

```

12 {
13     fps = frame;
14     frame = 0;
15     tCheck = tNow;
16 }
17
18 SelectObject(mdc,man[num]);
19 sprintf(str,"每秒钟显示 %d 个画面 ",fps);
20 TextOut(mdc,0,0,str,strlen(str));
21 BitBlt(hdc,0,0,600,450,mdc,0,0,SRCCOPY);
22
23 tPre = GetTickCount(); //记录此次绘图时间
24 num++;
25 }

```

### 程序说明

当游戏循环中判断窗口画面需要更新时，则调用 `MyPaint()` 函数进行下一张人物位图的贴图操作。

(1) 第 10 行：每次调用此函数则将画面更新次数累加“1”。

(2) 第 11~15 行：判断此次绘图时间由前一秒算起是否已经到达 1 秒钟的间隔。若是，则将目前的“frame”值赋给“fps”，表示这一秒内所更新的画面次数。然后将“frame”值归“0”，并重设下次计算每秒画面数的起始时间“tCheck”。

(3) 第 18~21 行：选用要更新的图案到 mdc 中，再输出显示每秒画面更新次数的字符串到 mdc 上，最后将 mdc 的内容贴到窗口中。

(4) 第 23 行：在这次画面更新完毕之后，记录更新的时间，以供下次游戏循环中判断是否已达到画面更新操作设定的时间间隔。

### 运行结果

程序运行结果如图 3-3 所示。



图 3-3

这个范例中设定画面更新的时间间隔是 0.1 秒，所以每秒钟最多会更新 10 次画面。不过如果在范例运行的同时运行其他程序，那么 CPU 必须马上去处理所开启的其他程序，因此可能会使得

每秒画面的更新次数稍稍下降。这样的情形在每秒画面更新次数越高时会越显著。

### 3.1.3 透明动画

“透明动画”是游戏中一定会用到的基本技巧，它通过图案的连续显示及透明来产生背景图上的动画效果。前面已介绍使用游戏循环显示动画的技巧，也介绍了位图透明的方式，这一小节中将用一个范例来说明透明动画的制作过程。

这个范例使用了如图 3-4 所示的恐龙跑动连续图，每一张跑动图片的宽高为  $95 \times 99$ 。透明动画制作的前提是，必须在一个暂存的内存 DC 上完成每一张跑动图的透明然后再贴到窗口上，这样在画面更新时才不会出现透明贴图过程中产生的闪烁现象。

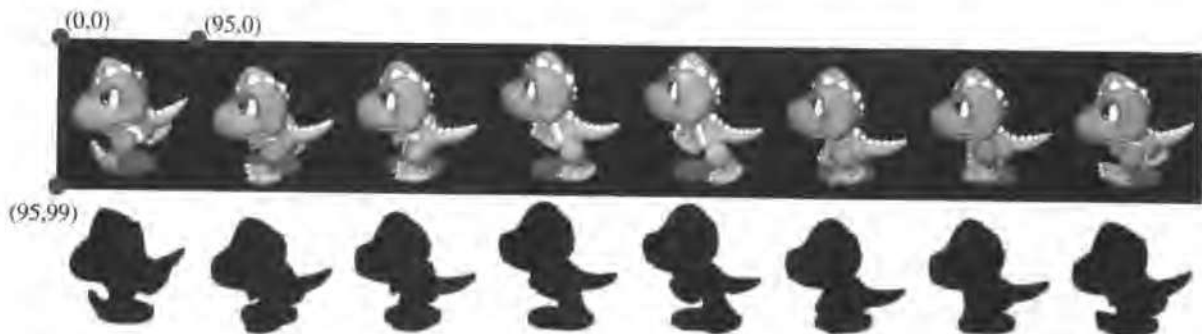


图 3-4

#### » 范例 ch3\_3

显示连续动态前景图案，并在显示之前进行透明，从而产生透明动画效果。

程序代码：全局变量声明

```
1 //全局变量声明
2 HINSTANCE hInst;
3 HBITMAP dra,bg;
4 HDC hdc,mhc,bufdc;
5 HWND hWnd;
6 DWORD tPre,tNow;
7 int num,x,y;
```

程序说明

- (1) 第 3 行：声明位图对象。“bg”为背景图，“dra”为恐龙连续跑动及对应的屏蔽图。
- (2) 第 4 行：声明 DC 对象。各 DC 对象用途说明如表 3-2 所示。

表 3-2

DC 名称	说明
hdc	窗口 DC
mhc	存储要贴到窗口上的内容。并在此内存 DC 上进行透明处理
bufdc	选取位图用的 DC

(3) 第 7 行:  $x$ 、 $y$  为每次恐龙图案的贴图坐标。利用贴图坐标值的改变, 产生动画前进的效果。

## 程序代码: InitInstance()

```

1  //****初始化函数*****
2  // 加载位图并设定各对象的初始值
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {
5      char filename[20] = "";
6      HBITMAP bmp;
7      hInst = hInstance;
8
9      hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
10         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
11
12      if (!hWnd)
13      {
14          return FALSE;
15      }
16
17      MoveWindow(hWnd, 10, 10, 640, 480, true);
18      ShowWindow(hWnd, nCmdShow);
19      UpdateWindow(hWnd);
20
21      hdc = GetDC(hWnd);
22      mdc = CreateCompatibleDC(hdc);
23      bufdc = CreateCompatibleDC(hdc);
24
25      bmp = CreateCompatibleBitmap(hdc, 640, 480);
26      SelectObject(mdc, bmp);
27
28      dra = (HBITMAP)LoadImage(NULL, "dra.bmp", IMAGE_BITMAP, 760, 198,
29         LR_LOADFROMFILE);
30
31      bg = (HBITMAP)LoadImage(NULL, "bg.bmp", IMAGE_BITMAP, 640, 480, LR_LOADFROMFILE);
32
33      num = 0;      //显示图号
34      x = 640;      //贴图起始 X 坐标
35      y = 360;      //贴图起始 Y 坐标
36
37      MyPaint(hdc);
38      return TRUE;
39  }

```

## 程序说明

(1) 第 25~26 行: 由于后面的程序是在 mdc 上进行透明处理的, 因此必须建立一个空的位图对象, 并置于 mdc 中。

(2) 第 28~33 行: 分别加载跑动的恐龙图及背景图, 并设定各项变量的初始值。

## 程序代码: MyPaint()

```

1  /***自定义绘图函数***/
2  // 1.恐龙跑动图案透明
3  // 2.更新贴图坐标
4  void MyPaint(HDC hdc)
5  {
6      if(num == 8)
7          num = 0;
8
9      //在mdc中贴上背景图
10     SelectObject(bufdc,bg);
11     BitBlt(mdc,0,0,640,480,bufdc,0,0,SRCCOPY);
12
13     //在mdc上进行透明处理
14     SelectObject(bufdc,dra);
15     BitBlt(mdc,x,y,95,99,bufdc,num*95,99,SRCAAND);
16     BitBlt(mdc,x,y,95,99,bufdc,num*95,0,SRCPAINT);
17
18     //将最后画面显示在窗口中
19     BitBlt(hdc,0,0,640,480,mdc,0,0,SRCCOPY);
20
21     tPre = GetTickCount(); //记录此次绘图时间
22     num++;
23
24     x-=20; //计算下次贴图坐标
25     if(x<=-95)
26         x = 640;
27 }

```

## 程序说明

(1) 第7~17行: 在mdc上进行透明操作并将最后结果显示在窗口中。

(2) 第13~14行: 进行透明时, 按照目前的图号来取出对应的跑动图案或者屏蔽图。下面以图3-5来说明。

num=0, 取出第0张图

左上点坐标(0×95,0)=(0,0)

num=3, 取出第3张图

左上点坐标(3×95,0)=(285,0)

num=6, 取出第6张图

左上点坐标(6×95,0)=(570,0)



图3-5

(3) 第22~24行: 计算下次恐龙图的贴图坐标。由于这个范例的动画是由右向左跑动的, 在Y轴坐标上不变化, 而X轴坐标每次向左递减20, 直到图案跑到左边窗口外时再将坐标设回最右边, 这样将可看到恐龙不断地由右向左循环跑动的效果。



### 运行结果

程序运行结果如图 3-6 所示。

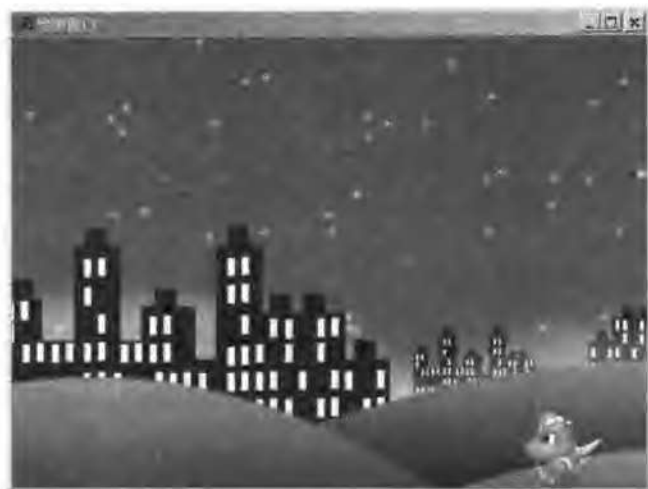


图 3-6 显示透明动画

## 3.2 动画显示问题

上一节里介绍了游戏动画的制作，而动画最基本的要求是画面要流畅且符合真实性。然而由于是利用贴图的方法来产生动画的，因此常会因一些小细节没注意到而使得动画的效果看起来不太自然。在这一节里就提出了两个这方面的小问题，在制作动画时若特别注意，可使得游戏动画看起来更加顺畅。

### 3.2.1 贴图坐标修正

动画的制作需要多张连续的图片，如果这些连续图片规格不一，那么进行贴图时就需要进行贴图坐标的修正，否则就可能产生动画晃动和不顺畅的情形。

对于有经验的游戏美术设计人员来说，是不会有图片规格本该一致却不一致的错误发生的，即使图片尺寸有误也会重新进行修改。但有时有一些无法避免的情形，使得必须在游戏程序中进行贴图坐标的修正。这里举个例子，请看如图 3-7 所示的这些恐龙上下左右跑动的连续图片。





图 3-7

可以看出，恐龙在同一方向上的跑动图案大小是一样的，而在不同方向上的尺寸却略有不同。这在动画贴图的时候会有一点小问题。现在假设图中这只恐龙的操作是，原本面向左然后变成面向下。恐龙本身并没有移动，那么程序会先贴面向左的图案再贴面向下的图案，如果这两个贴图操作都使用相同左上角的贴图坐标，那么产生的结果如图 3-8 所示。

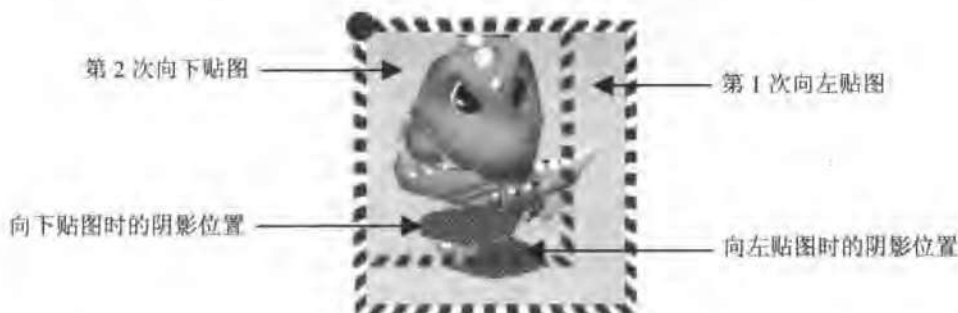


图 3-8

两次贴图，恐龙不过是做了个由左向下转的动作，但它的阴影所在位置竟然移动了，这也意味着恐龙在这一个动作之间产生了移动。事实上这是不对的，而这样的贴图方法会使得动画制作时产生误差。因此在这样的情况下，必须对贴图坐标进行修正，如图 3-9 所示。

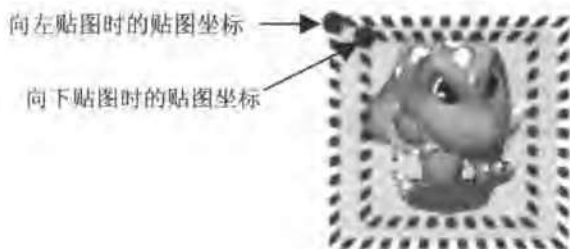


图 3-9

图中以阴影部分作为贴图的基准，在恐龙动作转而面向下时做贴图坐标的修正，使得第 2 次贴图时阴影部分能够与上一次重叠。由图中可看出，第 2 次贴图的左上角坐标与第 1 次相比稍微往右下方移动了，这样的修正使得动画显示时视觉效果更好。

这一小节里以恐龙图为例说明了动画贴图坐标的修正问题，下一小节里则要继续讨论另一个重要的动画贴图问题“排序贴图”，届时将会看到一个完整的恐龙跑动动画范例。

## 3.2.2 排序贴图

“排序贴图”的问题源自于物体远近呈现的一种贴图概念。回忆之前贴图的方式，对于距离较远的物体先进行贴图操作，然后再进行近距离物体的贴图操作，而一旦定出贴图的顺序之后就无法再改变了。这样的作法在画面上物体会彼此遮掩的情况下便不适用，下面以图 3-10 来做说明。

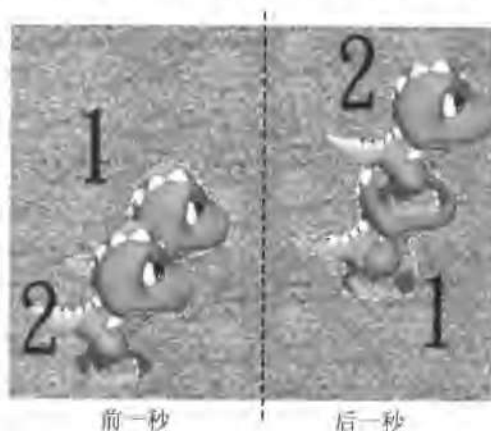


图 3-10

图中把两只恐龙做了编号，在这个例子里，先进行 1 号恐龙的贴图操作，然后再进行 2 号恐龙的贴图操作。在前一秒的画面里，我们可以看到画面还很正常，可是到了后一秒的时候，画面却怪怪的，这是因为此时的 2 号恐龙已经跑到了 1 号恐龙的后面，但是贴图顺序还是先贴 1 号恐龙图案再贴 2 号恐龙图案，变成了后面的物体反而遮掩住了前面的物体这种不协调的画面。

为了避免这种因为贴图顺序固定而产生的错误画面，必须在每一次窗口重新显示时动态地重新决定画面上每一个物体的贴图顺序。如何动态决定贴图顺序呢？这里采用的方法是“排序”。

排序如何运用在贴图中呢？这里举个例子，假设现在有 10 只要进行贴图的恐龙图案。先把它存在一个数组之中，从 2D 平面的远近角度来看，Y 轴坐标比较小（在窗口画面较上方）的是比较远的物体。如果我们以恐龙的 Y 轴坐标（在排序中称为键值）来对恐龙数组由小到大进行排序，最后会使得 Y 轴坐标小的恐龙排在数组的前面，而进行画面贴图时则由数组从小到大一个一个进行处理，这样便可实现“远的物体先贴图”的目的了。

要进行排序，首先必须决定所使用的排序法，这里所用的排序法为气泡排序（Bubble Sort），使用此排序方法的原因有下列 3 个。

- （1）程序代码简单。
- （2）排序效率中等。
- （3）属于稳定（stable）排序法，这个特性会使得 Y 轴坐标相同的物体，不必再去考虑它 X 坐标上的排序。

下面列出了以 C/C++ 语言写出的气泡排序法程序，其中对“Obj[]”数组的各元素以其数据成员的 y 值为键值来做排序，输入的参数为“n”表示要排序数组的大小：

```

1 void BubSort(int n)                //气泡排序
2 {
3     int i, j;
  
```

```

4   bool f;                                //测试旗标
5   Obj tmp;                               //数组元素交换时暂存用的变量
6
7   for(i=0;i<n-1;i++)
8   {
9       f = false;
10      for(j=0;j<n-i-1;j++)
11      {
12          if(Obj[j+1].y < Obj[j].y)
13          {
14              //进行数组元素交换
15              tmp = Obj[j+1];
16              Obj[j+1] = Obj[j];
17              Obj[j] = tmp;
18              f = true;
19          }
20      }
21      if(!f)                                //无交换操作则结束循环
22          break;
23  }
24  }

```

下面就气泡排序的原理来稍做说明：气泡排序法的步骤是从数组的最前面开始进行两两比较，当下一个元素值比前一个元素值大时则进行交换，使得大的元素往后移。这样不断进行比较（程序中的内层循环），较大的元素会一直往后挪，最后数组中的最大元素便会排在数组的最后。在排好了数组中的最大元素之后，接着按照同样的方法再从头部进行两两比较，第2次便会将数组中的第2大元素排在数组倒数第2个位置上，这样不断地循环比较（程序中的外层循环），当外层循环结束后，数组就成了由小到大的排列顺序了。

下面以一个大小为5的数字数组来示范气泡排序的大致过程。

第1次外层循环运行：

7   5   2   1   3	
└─┘	5跟7比，7比较大，两者交换
5   7   2   1   3	
└─┘	7跟2比，7比较大，两者交换
5   2   7   1   3	
└─┘	7跟1比，7比较大，两者交换
5   2   1   7   3	
└─┘	7跟3比，7比较大，两者交换
5   2   1   3   7	排好了数组最后一个元素

第2次外层循环运行：

5   2   1   3   7	
└─┘	5跟2比，5比较大，两者交换
2   5   1   3   7	
└─┘	5跟1比，5比较大，两者交换

2 1 5 3 7



5 跟 3 比, 5 比较大, 两者交换

2 1 3 5 7

排好了数组最后两个元素

第 3 次外层循环运行后的排列结果:

1 2 3 5 7

第 4 次外层循环运行时, 由于上次已经排序完成, 所以此次内层循环运行没有任何交换操作, 测试旗标的值是 false。内层循环运行后的 if 叙述会判断若无任何的交换操作发生则结束外层循环, 不过在此例中外层循环的运行次数刚好也是 4 次 ( $i=0$  到 3)。

以上就是对于气泡排序的一个大致说明, 您也可以再仔细研究这个程序详细的排序过程。接下来将以一个范例程序来演示气泡排序法在画面贴图上的运用, 让动画能呈现更接近真实的远近层次效果。

## » 范例 ch3\_4

产生多只恐龙随机跑动, 每次进行画面贴图前先完成排序操作, 并对恐龙跑动进行贴图坐标修正, 呈现较为顺畅真实的动画。

### 程序代码: 结构定义与常变量声明

```
1 //定义结构
2 struct dragon
3 {
4     int x,y;
5     int dir;
6 };
7
8 //常数定义与全局变量声明
9 const int draNum = 10;
10
11 HINSTANCE     hInst;
12 HBITMAP       draPic[4],bg;
13 HDC           hdc,mdc,bufdc;
14 HWND          hWnd;
15 DWORD         tPre,tNow;
16 int           picNum;
17 dragon        dra[draNum];
```

### 程序说明

(1) 第 1~6 行: 定义 dragon 结构, 代表画面上的恐龙, 其结构成员 x 和 y 为贴图的坐标, dir 为目前恐龙的移动方向。

(2) 第 9 行: 定义常数 draNum, 代表程序在画面上要出现的恐龙数目, 在此设定为 10。

(3) 第 12 行: draPic[4]存储恐龙上下左右移动的连续图案, bg 则存储背景图。

(4) 第 17 行: 按照 draNum 的值建立数组 dra[], 产生画面上出现的恐龙。

### 程序代码: InitInstance()

```
1 //****初始化函数*****
2 // 加载位图并设定各初始值
3 BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
```

```
4 {
5     HBITMAP bmp;
6     hInst = hInstance;
7     int i;
8
9     hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
10         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
11
12     if (!hWnd)
13     {
14         return FALSE;
15     }
16
17     MoveWindow(hWnd, 10, 10, 640, 480, true);
18     ShowWindow(hWnd, nCmdShow);
19     UpdateWindow(hWnd);
20
21     hdc = GetDC(hWnd);
22     mdc = CreateCompatibleDC(hdc);
23     bufdc = CreateCompatibleDC(hdc);
24
25     bmp = CreateCompatibleBitmap(hdc, 640, 480);
26     SelectObject(mdc, bmp);
27
28     draPic[0] = (HBITMAP) LoadImage(NULL, "dra0.bmp", IMAGE_BITMAP, 528, 188,
29         LR_LOADFROMFILE);
30     draPic[1] = (HBITMAP) LoadImage(NULL, "dra1.bmp", IMAGE_BITMAP, 544, 164,
31         LR_LOADFROMFILE);
32     draPic[2] = (HBITMAP) LoadImage(NULL, "dra2.bmp", IMAGE_BITMAP, 760, 198,
33         LR_LOADFROMFILE);
34     draPic[3] = (HBITMAP) LoadImage(NULL, "dra3.bmp", IMAGE_BITMAP, 760, 198,
35         LR_LOADFROMFILE);
36     bg = (HBITMAP) LoadImage(NULL, "bg.bmp", IMAGE_BITMAP, 640, 480, LR_LOADFROMFILE);
37
38     for(i=0; i<draNum; i++)
39     {
40         dra[i].dir = 3;           //起始方向
41         dra[i].x = 200;          //贴图的起始 x 坐标
42         dra[i].y = 200;          //贴图的起始 y 坐标
43     }
44
45     MyPaint(hdc);
46
47     return TRUE;
48 }
```

#### 程序说明

- (1) 第 25~26 行: 建立一个空位图并放入 mdc 中。
- (2) 第 28~32 行: 加载各张恐龙跑动图及背景图。本程序以 0、1、2、3 来代表恐龙的上、

下、左、右移动。

(3) 第 34 ~ 39 行: 设定所有恐龙初始的贴图坐标都为 (200,200), 初始的移动方向都为向左。

## 程序代码: BubSort()

```

1  //气泡排序法
2  void BubSort(int n)
3  {
4      int i,j;
5      bool f;
6      dragon tmp;
7
8      for(i=0;i<n-1;i++)
9      {
10         f = false;
11         for(j=0;j<n-i-1;j++)
12         {
13             if(dra[j+1].y < dra[j].y)
14             {
15                 tmp = dra[j+1];
16                 dra[j+1] = dra[j];
17                 dra[j] = tmp;
18                 f = true;
19             }
20         }
21         if(!f)
22             break;
23     }
24 }
  
```

## 程序说明

气泡排序的程序内容与前面说明时所看到的程序内容相同, 在这个范例中是对 dra[] 数组进行排序的。

## 程序代码: MyPaint()

```

1  //****自定义绘图函数*****
2  // 1.对窗口中跑动的恐龙进行排序贴图
3  // 2.恐龙贴图坐标修正
4  void MyPaint(HDC hdc)
5  {
6      int w,h,i;
7
8      if(picNum == 8)
9          picNum = 0;
10
11     //在mdc 中先贴上背景图
12     SelectObject(bufdc,bg);
13     BitBlt(mdc,0,0,640,480,bufdc,0,0,SRCCOPY);
14
15     BubSort(draNum);
  
```

```

16
17 for(i=0;i<draNum;i++)
18 {
19     SelectObject(bufdc, draPic[dra[i].dir]);
20     switch(dra[i].dir)
21     {
22         case 0:
23             w = 66;
24             h = 94;
25             break;
26         case 1:
27             w = 68;
28             h = 82;
29             break;
30         case 2:
31             w = 95;
32             h = 99;
33             break;
34         case 3:
35             w = 95;
36             h = 99;
37             break;
38     }
39     BitBlt(mdc, dra[i].x, dra[i].y, w, h, bufdc, picNum*w, h, SRCAND);
40     BitBlt(mdc, dra[i].x, dra[i].y, w, h, bufdc, picNum*w, 0, SRCPAINT);
41 }
42
43 //将最后画面显示在窗口中
44 BitBlt(hdc, 0, 0, 640, 480, mdc, 0, 0, SRCCOPY);
45
46 tPre = GetTickCount(); //记录此次绘图时间
47 picNum++;
48
49 for(i=0;i<draNum;i++)
50 {
51     switch(rand()%4) //随机决定下次移动方向
52     {
53         case 0: //上
54             switch(dra[i].dir)
55             {
56                 case 0:
57                     dra[i].y -= 20;
58                     break;
59                 case 1:
60                     dra[i].x += 2;
61                     dra[i].y -= 31;
62                     break;
63                 case 2:
64                     dra[i].x += 14;

```



```

65         dra[i].y -= 20;
66         break;
67     case 3:
68         dra[i].x += 14;
69         dra[i].y -= 20;
70         break;
71     }
72     if(dra[i].y < 0)
73         dra[i].y = 0;
74     dra[i].dir = 0;
75     break;
76 case 1:                                     //下
77     switch(dra[i].dir)
78     {
79         case 0:
80             dra[i].x -= 2;
81             dra[i].y += 31;
82             break;
83         case 1:
84             dra[i].y += 20;
85             break;
86         case 2:
87             dra[i].x += 15;
88             dra[i].y += 29;
89             break;
90         case 3:
91             dra[i].x += 15;
92             dra[i].y += 29;
93             break;
94     }
95     if(dra[i].y > 370)
96         dra[i].y = 370;
97     dra[i].dir = 1;
98     break;
99 case 2:                                     //左
100    switch(dra[i].dir)
101    {
102        case 0:
103            dra[i].x -= 34;
104            break;
105        case 1:
106            dra[i].x -= 34;
107            dra[i].y -= 9;
108            break;
109        case 2:
110            dra[i].x -= 20;
111            break;
112        case 3:
113            dra[i].x -= 20;

```

```

114             break;
115         }
116         if(dra[i].x < 0)
117             dra[i].x = 0;
118         dra[i].dir = 2;
119         break;
120     case 3: //右
121         switch(dra[i].dir)
122         {
123             case 0:
124                 dra[i].x += 6;
125                 break;
126             case 1:
127                 dra[i].x += 6;
128                 dra[i].y -= 10;
129                 break;
130             case 2:
131                 dra[i].x += 20;
132                 break;
133             case 3:
134                 dra[i].x += 20;
135                 break;
136         }
137         if(dra[i].x > 535)
138             dra[i].x = 535;
139         dra[i].dir = 3;
140         break;
141     }
142 }
143 }

```

#### 程序说明

MyPaint()函数可分成两部分,前半段先将数组中各恐龙按照目前所在的坐标进行排序贴图的操作;后半段则是随机决定下次恐龙的移动方向,并计算下次所有恐龙的贴图坐标,因此,每次调用此函数时,都会进行窗口画面的更新,产生恐龙四处移动的效果。

(1) 第 15 行:在贴上恐龙图之前调用 BubSort()函数进行排序。

(2) 第 17~41 行:按照目前恐龙的移动方向 dra[i].dir,选取对应的位图到 bufdc 中,并设定裁切的大小。将每一张要在窗口上出现的恐龙图案依次先在 mdc 上进行透明贴图操作。

(3) 第 44 行:将最后贴图完成的内容显示在窗口中。

(4) 第 49~143 行:决定每一只恐龙下一次的移动方向及贴图坐标。

(5) 第 51 行:按照随机数除 4 的余数来决定下次的移动方向,余数为 0、1、2、3 分别表示下次移动方向为向上、向下、向左、向右。

(6) 第 53~75 行:以下次移动方向向上为例,其中第 54~71 行按照目前的移动方向来修正因各方向上图案尺寸不一而产生的贴图坐标误差,加入恐龙每次移动的单位量(上、下、左、右每次移动量都为 20 个单位)而得到下次新的贴图坐标。

(7) 第 72~74 行:在计算出新的贴图坐标之后,还必须判断此新的坐标会不会使得恐龙贴图

超出窗口边界。若会超出，则将该方向上的坐标设定为刚好等于临界值。

(8) 第 76~141 行：其他方向上的移动量按照相同的方法计算。

## 运行结果

程序运行结果如图 3-11 所示。



图 3-11

从图中可以看出，由于在贴图前先进行了排序操作，因此使得恐龙彼此之间没有遮掩。可以通过设定程序中最前面定义的“draNum”常数值来改变画面上出现的恐龙数目。

## 3.3 背景动画设计

前面的内容中介绍了前景动画的设计，接下来是背景动画的设计制作。背景动画的制作同样是利用连续贴图的原理。2D 游戏中经常运用到的动态背景表现手法大致有 3 种，这一节中就将这 3 种游戏中的背景动画模式进行介绍。

### 3.3.1 单一背景滚动

单一背景滚动的方法是：利用一张相当大的背景图，当游戏进行的时候，随着画面中人物的移动，背景的显示区域也跟着移动。要制作这样的背景滚动效果实际上很简单，只要在每次背景画面更新时改变要显示到窗口上的区域就可以了，如图 3-12 所示。

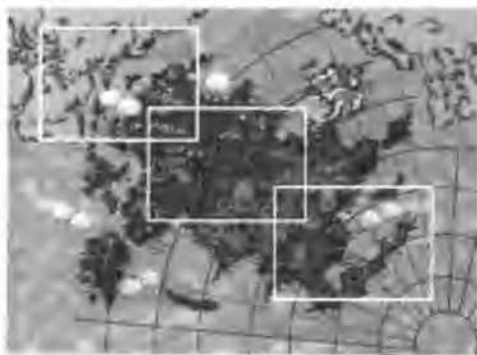


图 3-12

例如在上面的这张背景图里,由左上到右下图了3个方框,代表要显示在窗口上的背景区域,程序只要按照左上到右下的顺序在窗口上连续显示这3个方框区域,就会产生背景由左上往右下滚动的效果。

背景滚动的概念并不难理解,下面直接以一个范例来看看单一背景滚动的制作方法。

## » 范例 ch3\_5

以键盘【↑】、【↓】、【←】、【→】键控制背景滚动显示。

### 程序代码: 全局变量声明

```
1 //全局变量声明
2 HINSTANCE hInst;
3 HBITMAP map;
4 HDC hdc,mdc;
5 HWND hWnd;
6 DWORD tPre,tNow;
7 int x=0,y=0;
```

### 程序说明

第7行: 声明两变量  $x$ 、 $y$  分别代表背景图要剪裁的左上角点坐标,初始值设为 (0,0),表示背景第一次显示时是在最左上角。

### 程序代码: WndProc()

```
1 //****消息处理函数*****
2 // 根据按下的方向键改变背景图剪裁的左上角点的坐标
3 LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
  lParam)
4 {
5     switch (message)
6     {
7         case WM_KEYDOWN: //按下键盘消息
8             switch (wParam)
9             {
10                case VK_UP: //按下【↑】键
11                    y -= 20;
12                    if(y < 0)
13                        y = 0;
14                    break;
15                case VK_DOWN: //按下【↓】键
16                    y += 20;
17                    if(y > 660)
18                        y = 660;
19                    break;
20                case VK_LEFT: //按下【←】键
21                    x -= 20;
22                    if(x < 0)
23                        x = 0;
24                    break;
25                case VK_RIGHT: //按下【→】键
```

```

26             x += 20;
27             if(x > 910)
28                 x = 910;
29             break;
30         }
31         break;
32     case WM_DESTROY:                //窗口结束消息
33         DeleteDC(mdc);
34         DeleteObject(map);
35         ReleaseDC(hWnd, hdc);
36         PostQuitMessage(0);
37         break;
38     default:                        //其他消息
39         return DefWindowProc(hWnd, message, wParam, lParam);
40 }
41 return 0;
42 }

```

## 程序说明

在 WndProc() 消息处理函数中加入 “WM\_KEYDOWN” 代表按下键盘消息的处理，当使用者按下【↑】、【↓】、【←】、【→】键时，背景就往对应的方向滚动 20 个单位。

(1) 第 10~13 行：以按下【↑】键的情况来做说明：“VK\_UP” 是代表按下【↑】键对应的伪码（下一章会详细介绍），当【↑】键按下时，则将 “y” 值减 20，即下次截取背景的左上角点坐标往上移 20 个单位，这样再次贴图时便会产生背景图往上移动的效果。

(2) 第 15~29 行：当按下其他方向的按键时，则按照相同的方法来增减对应方向上截取背景左上点的坐标值。

## 程序代码：MyPaint()

```

1  //****自定义绘图函数*****
2  // 按照背景图剪裁的左上角点坐标进行贴图
3  void MyPaint(HDC hdc)
4  {
5      BitBlt(hdc, 0, 0, 640, 480, mdc, x, y, SRCCOPY);
6      tPre = GetTickCount();
7  }

```

## 程序说明

MyPaint() 函数中以目前的 x、y 值为截取背景图区域的左上点坐标。当使用者按下方向键时，这里的 x、y 值就会跟着改变，而当游戏循环调用 MyPaint() 函数进行绘图时，就会产生背景移动的效果。

## 运行结果

程序运行结果如图 3-13 所示。



图 3-13

### 3.3.2 循环背景动画

循环背景是不断地进行背景图的裁切与接合,然后显示在窗口上所产生的的一种背景画面循环滚动的效果。下面就介绍如何利用同一张跟窗口大小相同的背景天空图案来产生背景由左向右循环滚动的动画效果的。

首先来介绍背景图由左向右滚动的概念。假设如图 3-14 所示的这张图是前一秒画面更新时所看到的画面(外围的方框代表窗口),当下一秒背景向右滚动时,则显示画面如图 3-15 所示。



图 3-14

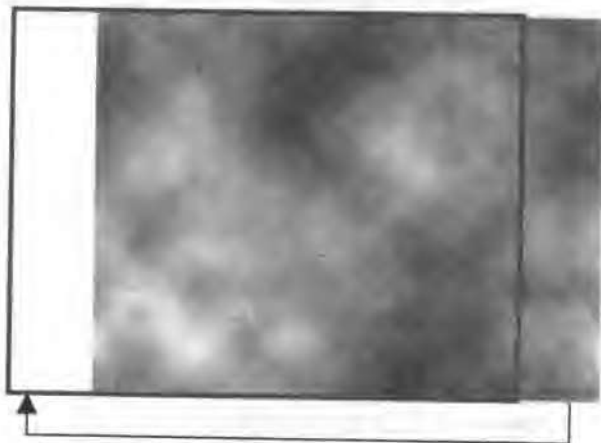


图 3-15

由图中可以看出,窗口中背景图部分跟前一次相比已经往右移动了。在制作循环背景的过程中,是把超出窗口的部分贴到左边空出的窗口区域中,以便重新组合成一张刚好等于窗口大小的新背景图。

上面的背景图滚动的概念可以很容易地以两次贴图的方式来完成,下面以图示来做说明。在此假设最原始的背景图已经被放入一个 DC 对象当中,背景图的尺寸为  $640 \times 480$  且刚好与窗口大小相同。将在另一 DC 对象上来完成背景图两次贴图的操作。

**步骤一:** 截取原始背景图右边部分进行贴图操作到另一 DC 中,假设目前要裁取的右边部分的宽度为  $x$ ,如图 3-16 所示。

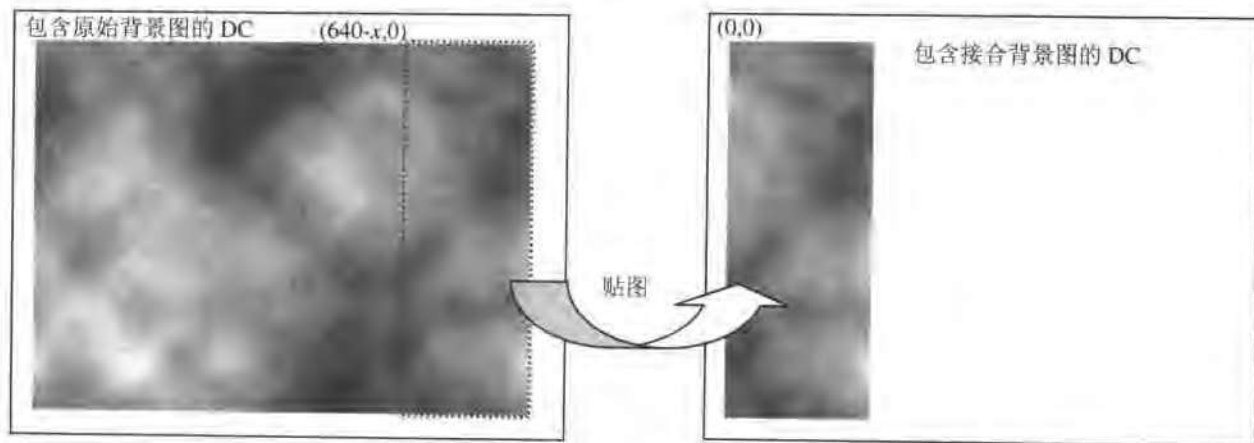


图 3-16

步骤二：裁取原始背景图左边部分进行贴图操作到另一 DC 中，这样就完成了向右滚动接合后的新背景图，如图 3-17 所示。

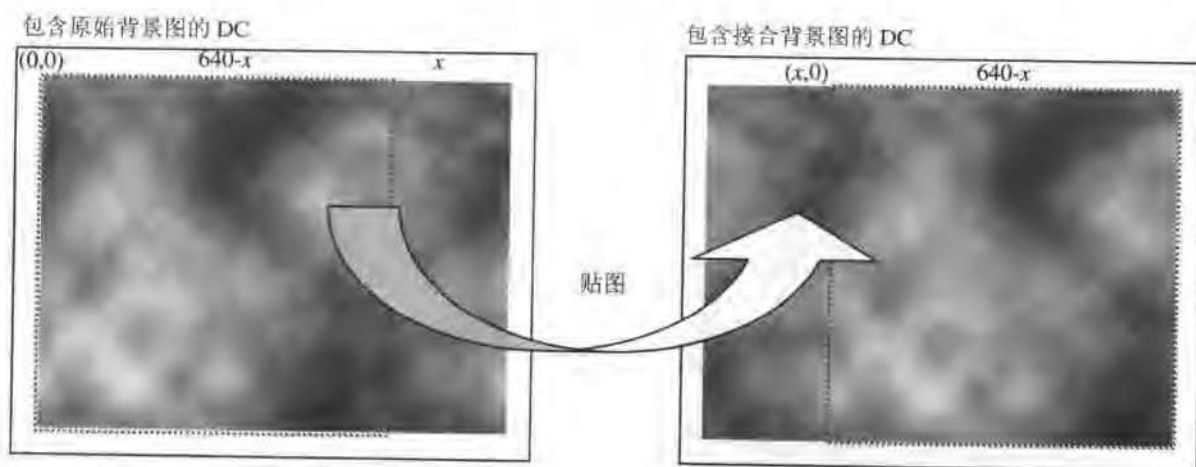


图 3-17

步骤三：将接合后的背景图显示在窗口中，之后递增  $x$  值。重复步骤一、步骤二和步骤三来产生背景图慢慢向右滚动的效果。当  $x$  值递增到大于或等于背景图的大小时，则将  $x$  的值重设为“0”，这样就会产生背景循环的效果。

这里要提醒注意的是，若要制作像上面这样的背景循环动画，所使用的背景图必须是接合后看不出图案接缝的图，否则产生的循环画面就不完美了。明白了循环背景动画制作的原理之后，下面以一个范例来看看实际的制作方法。

## 》》范例 ch3\_6

利用一张  $640 \times 480$  的背景图，制作背景由左向右循环滚动的动画。

程序代码：全局变量声明

```

1  //全局变量声明
2  HINSTANCE hInst;
```

```

3  HBITMAP      bg;
4  HDC           hdc,mdc,bufdc;
5  HWND          hwnd;
6  DWORD         tPre,tNow;
7  int           x=0;

```

**程序说明**

第7行：变量“x”用来记录要裁取的右边部分的宽度，初始值设为“0”。

**程序代码：InitInstance()**

```

1  //****初始函数*****
2  // 加载点载图并将位图置入对应DC 中
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {
5      HBITMAP bmp;
6      hInst = hInstance;
7
8      hwnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
9          CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
10
11     if (!hwnd)
12     {
13         return FALSE;
14     }
15
16     MoveWindow(hwnd,10,10,640,480,true);
17     ShowWindow(hwnd, nCmdShow);
18     UpdateWindow(hwnd);
19
20     hdc = GetDC(hwnd);
21     mdc = CreateCompatibleDC(hdc);
22     bufdc = CreateCompatibleDC(hdc);
23
24     bmp = CreateCompatibleBitmap(hdc,640,480);
25     SelectObject(mdc,bmp);
26
27     bg = (HBITMAP)LoadImage(NULL,"bg.bmp", IMAGE_BITMAP,648,480,LR_LOADFROMFILE);
28     SelectObject(bufdc,bg);
29
30     MyPaint(hdc);
31
32     return TRUE;
33 }

```

**程序说明**

在此范例中，将原始背景图置入 bufdc 中，并在 mdc 中进行切割组合的贴图操作。

(1) 第24~25行：建立一个空的位图并置入 mdc 中。

(2) 第27~28行：载入背景图并置入 bufdc 中。



### 程序代码: MyPaint()

```

1  //****自定义绘图函数*****
2  // 切割与接合背景图产生循环背景
3  void MyPaint(HDC hdc)
4  {
5      //截取背景图右边部分进行贴图
6      BitBlt(mdc,0,0,x,480,bufdc,640-x,0,SRCCOPY);
7
8      //截取背景图左边部分进行贴图
9      BitBlt(mdc,x,0,640-x,480,bufdc,0,0,SRCCOPY);
10
11     //将接合后的背景图贴到窗口中
12     BitBlt(hdc,0,0,640,480,mdc,0,0,SRCCOPY);
13
14     tPre = GetTickCount();
15
16     x += 10;
17     if(x==640)
18         x = 0;
19 }

```

### 程序说明

每次调用 MyPaint()函数时,都会进行背景图切割接合,并在画面上显示背景图。此处利用 BitBlt()函数进行 3 次贴图来完成。

- (1) 第 6 行: 截取原始背景图右边的部分贴到 mdc 中。
- (2) 第 9 行: 截取原始背景图左边的部分贴到 mdc 中。
- (3) 第 12 行: 将前面两个步骤在 mdc 中完成的背景图显示在窗口中。
- (4) 第 16~18 行: 重设下次背景图右边区块要切割的宽度,在这个范例里每次累加 10 个单位,背景图每次便向右滚动 10 个单位显示。当要切割的宽度达到位图的宽度 640 时,则将 x 的值设为“0”,产生循环背景动画效果。

### 运行结果

程序运行结果如图 3-18 所示。



图 3-18 背景由左向右循环滚动

### 3.3.3 多背景循环动画

多背景循环动画的背景循环原理其实与前一小节讲过的背景循环的原理相同。不过由于不同背景在远近层次上及实际视觉移动速度上并不相同,因此在以贴图的方法制作多背景循环动画时,需要决定不同背景贴图的先后顺序及滚动的速度。

图3-19是这一小节里多背景循环动画范例的运行结果,画面中出现了几种背景及恐龙跑动的前景图。



图 3-19

观察上面的这张图,先要决定构成这幅画面的贴图顺序。从远近层次来看,天空是最远的,接着是草地和山峦,因为山峦叠在草地上,接下来是房屋,最后才是前景的恐龙,所以进行画面贴图时顺序应该是:

天空→草地→山峦→房屋→恐龙

另外,进行山峦、房屋及恐龙的贴图操作时,还需要进行透明的操作,才能使得这些物体能叠在它们前一层的背景上。

决定了贴图时的顺序之后,接着要来决定背景滚动时的速度。由于最远的背景是天空,所以当前景的恐龙跑动时,滚动速度应该是最慢的,而天空前的山峦滚动速度应该比天空要快一点,至于房屋与草地,因为连在一起,所以滚动速度相同,而且又会比山峦还要快一点,这样我们就决定出了所有背景的滚动速度为:

天空<山峦<草地=房屋

前景的恐龙只让它在原地跑动,由于背景自动向右滚动,因此就会产生恐龙向前奔跑的视觉效果。

接下来看看这样一个多背景循环动画的范例程序内容。

#### 》》范例 ch3\_7

运用贴图技巧并调整不同背景循环滚动的速度,显示具有远近层次感的多背景循环动画。

### 程序代码：全局变量声明

```

1  //全局变量声明
2  HINSTANCE  hInst;
3  HBITMAP    dra,bg[3];
4  HDC        hdc,mdc,bufdc;
5  HWND       hWnd;
6  DWORD      tPre,tNow;
7  int         x0=0,x1=0,x2=0,num=0;

```

### 程序说明

(1) 第3行：声明数组 bg[]用来存储各背景图，dra 存储前景的恐龙图。其中将天空和草地合并为同一张 640×480 的位图，如图 3-20 所示。



图 3-20

(2) 第7行：声明 x0、x1、x2，分别为各背景由左向右滚动时所要切割的右边区域的宽度。

### 程序代码：InitInstance()

```

1  //****初始函数*****
2  // 加载各位图并将建立的空位置置入 mdc 中
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {
5      HBITMAP bmp;
6      hInst = hInstance;
7
8      hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
9          CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
10
11     if (!hWnd)
12     {
13         return FALSE;
14     }
15
16     MoveWindow(hWnd,10,10,640,480,true);
17     ShowWindow(hWnd, nCmdShow);

```

```

18 UpdateWindow(hWnd);
19
20 hdc = GetDC(hWnd);
21 mdc = CreateCompatibleDC(hdc);
22 bufdc = CreateCompatibleDC(hdc);
23
24 bmp = CreateCompatibleBitmap(hdc, 640, 480);
25 SelectObject(mdc, bmp);
26
27 bg[0] = (HBITMAP)LoadImage(NULL, "bg0.bmp", IMAGE_BITMAP, 640, 480,
    LR_LOADFROMFILE);
28 bg[1] = (HBITMAP)LoadImage(NULL, "bg1.bmp", IMAGE_BITMAP, 640, 600,
    LR_LOADFROMFILE);
29 bg[2] = (HBITMAP)LoadImage(NULL, "bg2.bmp", IMAGE_BITMAP, 640, 600,
    LR_LOADFROMFILE);
30 dra = (HBITMAP)LoadImage(NULL, "dra.bmp", IMAGE_BITMAP, 760, 198,
    LR_LOADFROMFILE);
31
32 MyPaint(hdc);
33
34 return TRUE;
35 }

```

#### 程序说明

- (1) 第 24~25 行: 建立一个空位图并置入 mdc 中。
- (2) 第 27~30 行: 载入各背景图及前景图。

#### 程序代码: MyPaint()

```

1  //****自定义绘图函数*****
2  // 1.按照各背景远近顺序进行循环背景贴图
3  // 2.进行前景恐龙图的透明贴图
4  // 3.重设各背景图的切割宽度与跑动恐龙图的图号
5  void MyPaint(HDC hdc)
6  {
7      //贴上天空图
8      SelectObject(bufdc, bg[0]);
9      BitBlt(mdc, 0, 0, x0, 300, bufdc, 640-x0, 0, SRCCOPY);
10     BitBlt(mdc, x0, 0, 640-x0, 300, bufdc, 0, 0, SRCCOPY);
11
12     //贴上草地图
13     BitBlt(mdc, 0, 300, x2, 180, bufdc, 640-x2, 300, SRCCOPY);
14     BitBlt(mdc, x2, 300, 640-x2, 180, bufdc, 0, 300, SRCCOPY);
15
16     //贴上山峦图并透明
17     SelectObject(bufdc, bg[1]);
18     BitBlt(mdc, 0, 0, x1, 300, bufdc, 640-x1, 300, SRCAND);
19     BitBlt(mdc, x1, 0, 640-x1, 300, bufdc, 0, 300, SRCAND);
20     BitBlt(mdc, 0, 0, x1, 300, bufdc, 640-x1, 0, SRCPAINT);
21     BitBlt(mdc, x1, 0, 640-x1, 300, bufdc, 0, 0, SRCPAINT);

```

```

22
23 //贴上房屋图并透明
24 SelectObject(bufdc,bg[2]);
25 BitBlt(mdc,0,250,x2,300,bufdc,640-x2,300,SRCAUD);
26 BitBlt(mdc,x2,250,640-x2,300,bufdc,0,300,SRCAUD);
27 BitBlt(mdc,0,250,x2,300,bufdc,640-x2,0,SRCPAINT);
28 BitBlt(mdc,x2,250,640-x2,300,bufdc,0,0,SRCPAINT);
29
30 //贴上恐龙图并透明
31 SelectObject(bufdc,dra);
32 BitBlt(mdc,250,350,95,99,bufdc,num*95,99,SRCAUD);
33 BitBlt(mdc,250,350,95,99,bufdc,num*95,0,SRCPAINT);
34
35 BitBlt(hdc,0,0,640,480,mdc,0,0,SRCCOPY);
36
37 tPre = GetTickCount();
38
39 x0 += 5; //重设天空背景切割宽度
40 if(x0==640)
41     x0 = 0;
42
43 x1 += 8; //重设山峦背景切割宽度
44 if(x1==640)
45     x1 = 0;
46
47 x2 += 16; //重设草地及房屋背景切割宽度
48 if(x2==640)
49     x2 = 0;
50
51 num++; //重设跑动图的图号
52 if(num == 8)
53     num = 0;
54 }

```

## 程序说明

每次调用 MyPaint()函数时,先在 mdc 上完成所有画面图案的贴图操作,然后显示到窗口上,最后重设下次各背景图的切割宽度及前景图的跑动图的编号。

(1) 第 8~10 行:按照 x0 的值先在 mdc 上进行天空背景的贴图。

(2) 第 13~14 行:草地背景滚动的速度跟房屋相同,因此按照 x2 的值在 mdc 上进行贴图。

(3) 第 17~21 行:按照 x1 的值在 mdc 上进行山峦背景的透明贴图。

(4) 第 24~28 行:按照 x2 的值在 mdc 上进行房屋背景的透明贴图。

(5) 第 31~33 行:完成所有背景的贴图之后,最后再进行前景恐龙图的透明贴图。

(6) 第 39~49 行:重新设定下次各背景图向右滚动所要切割的区域大小。由于前面已讨论过各背景的滚动速度为:天空<山峦<草地=房屋,因此每次画面更新之后,将 x0、x1 和 x2 分别递增 5、8 和 16,这样下次显示时,天空、山峦、房屋与草地会分别向右滚动 5、8、16 个单位,从而达到所希望的远近背景不同滚动速度的动画效果。

**运行结果**

程序运行结果如图 3-21 所示。



图 3-21 多背景循环动画展示

至此已介绍了如何制作游戏中显示的图形及动画，但是就玩家与游戏之间的互动而言似乎还缺少了什么。那就是对游戏的操作输入系统。下一章将就 Windows 系统下的键盘及鼠标消息的概念与处理方法进行介绍，并说明如何将其运用在游戏程序之中。

## 课后重点整理

- 游戏中显示动画的方法有两种：一种是直接播放影片文件（如 AVI 和 MPEG 文件），常用在游戏的片头与片尾；另一种则是游戏进行时利用连续贴图的方式，制造动画的效果。
- 定时器（Timer）对象可以每隔一段时间发出一个时间消息。程序一旦接收到这一消息，便可以决定接下来要做哪些事情。
- Windows API 的 SetTimer() 函数可为窗口建立一个定时器。
- 一旦建立了一个定时器，它将一直自动地按照设定的时间间隔发出 WM\_TIMER 消息。如果要停用某个定时器，必须使用 KillTimer() 函数。
- 游戏本身需要显示顺畅的游戏画面，使玩家感觉不到延迟的状况。要达到这一点，基本上游戏画面必须在一秒钟之内更新至少 25 次以上。
- 游戏循环是将原先程序中的消息循环加以修改，方法是判断其中的内容目前是否有要处理的消息。如果有则进行处理，否则按照设定的时间间隔来重绘画面。
- “透明动画”是游戏中一定会运用到的基本技巧，它通过图案的连续显示及透明来产生背景图上的动画效果。
- “排序贴图”的问题是源自于物体远近呈现的一种贴图概念。
- 背景动画的制作同样是利用连续贴图的原理。2D 游戏中经常运用到的动态背景表现手法

大致有 3 种：单一背景滚动、循环背景动画及多背景循环动画。

- 单一背景滚动的方法是：利用一张相当大的背景图，当游戏进行的时候，随着画面中人物的移动，背景的显示区域也跟着移动。
- 循环背景动画是不断地进行背景图的裁切与接合，然后显示在窗口上所产生的一种背景画面循环滚动的效果。
- 以贴图的方法制作多背景循环动画时，需要决定不同背景贴图的先后顺序及滚动的速度。

### 课后练习

1. 动画最基本的要求就是画面要流畅且符合真实性。然而由于是利用贴图的方法来产生动画的，因此常会因忽略一些小细节而使得动画的效果看起来不太自然。请问有什么改善方案？
2. 请说明如何将排序方法运用在贴图中？
3. 在排序贴图中采用的排序法为气泡排序（Bubble Sort）。请问使用此方法的主要原因为何？

# 第 4 章 游戏输入消息处理

## 4.1 键盘输入消息

键盘是计算机的标准配件之一，也是基本的程序输入装置。虽然目前大多数游戏程序以鼠标或者是游戏杆来作为主要的输入设备。相比之下，以键盘操纵游戏的方式似乎变得不太重要，但有时候还是会使用键盘来设计一些快速键的功能，例如跳过片头动画、结束游戏等。

接下来将针对 Windows 系统下键盘的基本概念及键盘消息的处理方式做一简要的介绍。

### 4.1.1 关于 Windows 中的键盘

在 Windows 系统中，对于键盘及其输入处理的方式有一些特别的定义。这一小节先来介绍键盘在 Windows 系统下的特性。

#### 1. 虚拟键码

在早期非 Windows 系统的机器上，如果一般应用程序要取得使用者键盘的输入，就必须去取得输入按键的“扫描码 (Scan code)”。扫描码由实际的键盘硬件所产生，但是由于不同国家不同区域所使用的键盘设备可能不同，因此 Windows 系统便发展出一套标准。其解决方法就是对所有键盘的按键定义出一组通用的“虚拟键码”，也就是说在 Windows 系统下所有的按键都会被视为虚拟键（包含鼠标键在内），而每一个虚拟键都有其对应的一个虚拟键码。

#### 2. 键盘消息

Windows 系统是一个消息驱动的环境，一旦使用者在键盘上进行输入操作，那么系统便会接收到对应的键盘消息。表 4-1 中列出了最常见的 3 种键盘消息。

表 4-1

消息代号	说 明
WM_KEYDOWN	按下按键消息
WM_KEYUP	松开按键消息
WM_CHAR	字符消息

当某一按键被按下时，伴随着这个操作所产生的是以虚拟键码类型传送的 WM\_KEYDOWN 与 WM\_KEYUP 消息。当程序接收到这些消息时，便可由虚拟键码的信息来得知是哪个按键被按下。

此外，WM\_CHAR 则是当按下的按键为定义于 ASCII 中的可打印字符时，便发出此字符消息。前面介绍的消息循环中的 Translate()函数的作用便是当按下的按键为可打印字符时，便将虚拟键码消息进行转换并且发出字符消息。



## 3. 系统键

Windows 系统本身定义了一组“系统键”，这些按键通常是【Alt】与其他按键的组合。系统键对于 Windows 系统本身有一些特定的作用，Windows 中也特别针对系统键定出了表 4-2 中的相关消息。

表 4-2

消息代号	说 明
WM_SYSKEYDOWN	按下系统键消息
WM_SYSKEYUP	松开系统键消息

消息代号中加入的“SYS”代表系统键按下消息，然而实际上程序中很少处理系统键消息，因为当这类消息发生时 Windows 会自行处理并进行相应的工作。

以上便是键盘在 Windows 系统下关于其定义及输入处理的一些基本概念。下一小节将继续介绍程序中键盘消息的实际处理方式。

## 4.1.2 键盘消息处理

键盘消息同样是在消息处理函数中来加以定义处理的，按下按键事件一定会紧随着一个松开按键的事件，因此 WM\_KEYDOWN 与 WM\_KEYUP 两种消息必定是成对发生的。但通常仅在程序中对 WM\_KEYDOWN 消息进行处理，而忽略 WM\_KEYUP 消息。

观察消息处理函数中所输入的两个参数 wParam 与 lParam：

```
LRESULT CALLBACK WndProc(HWND hWnd,
                          UINT message,
                          WPARAM wParam,
                          LPARAM lParam)
```

当键盘消息触发时，wParam 的值为按下按键的虚拟键码。Windows 中所定义的虚拟键码是以“VK\_”开头的。lParam 则存储按键的相关状态信息。因此，如果程序要对使用者的键盘输入操作进行处理，那么消息处理函数的内容可以定义如下。

```
1  LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
    lParam)
2  {
3      switch (message)
4      {
5          case WM_KEYDOWN:                //按下键盘消息
6              switch (wParam)
7              {
8                  case VK_ESCAPE:          //按下【Esc】键
9                      //定义处理程序
10                     break;
11                 case VK_UP:              //按下【↑】键
12                     //定义处理程序
13                     break;
14                 :
```

```

15         :
16     }
17     break;
18     case WM_DESTROY:                //窗口结束消息
19         PostQuitMessage(0);
20         break;
21     default:                        //其他消息
22         return DefWindowProc(hWnd, message, wParam, lParam);
23 }
24 return 0;
25 }

```

针对这个消息处理函数中键盘消息处理的程序部分说明如下。

(1) 第5行：定义处理“WM\_KEYDOWN”消息。

(2) 第6行：以“switch”叙述判断“wParam”的值来得知哪个按键被按下，并运行对应“case”中的按键消息处理程序。

从以上的说明可了解程序中处理键盘消息的方式。下面的范例是让使用者以【↑】、【↓】、【←】、【→】键进行输入，控制画面中人物的移动。这里使用了人物在4个不同方向上走动的连续图案，如图4-1所示。

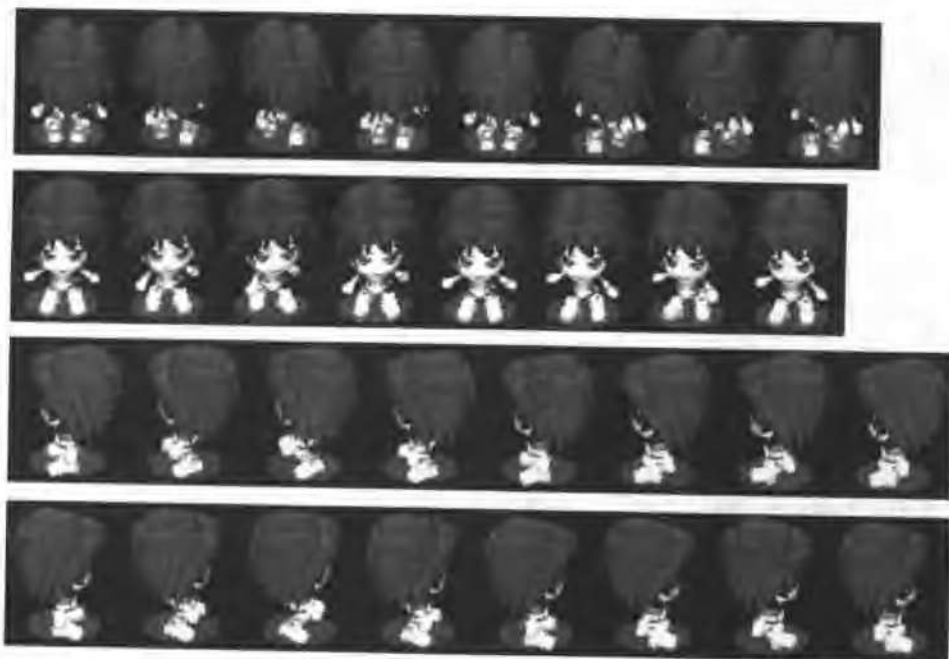


图 4-1

## » 范例 ch4\_1

处理键盘【↑】、【↓】、【←】、【→】键的输入消息，控制画面人物的移动。

程序代码：全局变量声明

```

1 //全局变量声明
2 HINSTANCE hInst;

```

```

3  HBITMAP girl[4],bg;
4  HDC      hdc,mdc,bufdc;
5  HWND hWnd;
6  DWORD    tPre,tNow;
7  int      num,dir,x,y;

```

## 程序说明

第7行：声明“x”、“y”变量为人物贴图坐标；“dir”变量为人物移动方向，范例中以“0”、“1”、“2”和“3”代表人物上、下、左、右方向上的移动；“num”变量则是连续图中的小图编号。

## 程序代码：InitInstance()

```

1  //****初始化函数*****
2  // 加载位图并设定各初始值
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {
5      HBITMAP bmp;
6      hInst = hInstance;
7
8      hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
9          CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
10
11     if (!hWnd)
12     {
13         return FALSE;
14     }
15
16     MoveWindow(hWnd,10,10,640,480,true);
17     ShowWindow(hWnd, nCmdShow);
18     UpdateWindow(hWnd);
19
20     hdc = GetDC(hWnd);
21     mdc = CreateCompatibleDC(hdc);
22     bufdc = CreateCompatibleDC(hdc);
23
24     bmp = CreateCompatibleBitmap(hdc,640,480);
25     SelectObject(mdc,bmp);
26
27     x = 300;
28     y = 250;
29     dir = 0;
30     num = 0;
31
32     girl[0] = (HBITMAP)LoadImage(NULL,"girl0.bmp",IMAGE_BITMAP,440,148,
33         LR_LOADFROMFILE);
34     girl[1] = (HBITMAP)LoadImage(NULL,"girl1.bmp",IMAGE_BITMAP,424,154,
35         LR_LOADFROMFILE);
36     girl[2] = (HBITMAP)LoadImage(NULL,"girl2.bmp",IMAGE_BITMAP,480,148,
37         LR_LOADFROMFILE);
38     girl[3] = (HBITMAP)LoadImage(NULL,"girl3.bmp",IMAGE_BITMAP,480,148,

```

```

        LR_LOADFROMFILE);
36  bg = (HBITMAP)LoadImage(NULL, "bg.bmp", IMAGE_BITMAP, 640, 480, LR_LOADFROMFILE);
37
38  MyPaint(hdc);
39
40  return TRUE;
41 )

```

#### 程序说明

- (1) 第24~25行：建立空的位图并置入 mdc 中。
- (2) 第27~29行：设定人物贴图初始坐标为 (0,0)，移动方向为向上。
- (3) 第32~36行：载入各连续移动位图及背景图。

#### 程序代码：WndProc

```

1  /****消息处理函数***/
2  // 1.按下【Esc】键结束程序
3  // 2.按下方向键重设贴图坐标
4  LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
    lParam)
5  {
6      switch (message)
7      {
8          case WM_KEYDOWN:                // 按键按下消息
9              switch (wParam)
10             {
11                 case VK_ESCAPE:          // 按下【Esc】键
12                     PostQuitMessage( 0 );
13                     break;
14                 case VK_UP:              // 按下【↑】键
15                     switch(dir)
16                     {
17                         case 0:
18                             y -= 10;
19                             break;
20                         case 1:
21                             x -= 1;
22                             y -= 8;
23                             break;
24                         case 2:
25                             x += 2;
26                             y -= 10;
27                             break;
28                         case 3:
29                             x += 2;
30                             y -= 10;
31                             break;
32                     }
33                     if(y < 0)
34                         y = 0;

```

```
35         dir = 0;
36         break;
37     case VK_DOWN:           //按下【↓】键
38         switch(dir)
39         {
40             case 0:
41                 x += 1;
42                 y += 8;
43                 break;
44             case 1:
45                 y += 10;
46                 break;
47             case 2:
48                 x += 3;
49                 y += 6;
50                 break;
51             case 3:
52                 x += 3;
53                 y += 6;
54                 break;
55         }
56
57         if(y > 375)
58             y = 375;
59         dir = 1;
60         break;
61     case VK_LEFT:           //按下【←】键
62         switch(dir)
63         {
64             case 0:
65                 x -= 12;
66                 break;
67             case 1:
68                 x -= 13;
69                 y += 4;
70                 break;
71             case 2:
72                 x -= 10;
73                 break;
74             case 3:
75                 x -= 10;
76                 break;
77         }
78         if(x < 0)
79             x = 0;
80         dir = 2;
81         break;
82     case VK_RIGHT:          //按下【→】键
83         switch(dir)
```

```

84         {
85             case 0:
86                 x += 8;
87                 break;
88             case 1:
89                 x += 7;
90                 y += 4;
91                 break;
92             case 2:
93                 x += 10;
94                 break;
95             case 3:
96                 x += 10;
97                 break;
98         }
99         if(x > 575)
100             x = 575;
101         dir = 3;
102         break;
103     }
104     break;
105 case WM_DESTROY:                                //窗口结束消息
106     int i;
107
108     DeleteDC(mdc);
109     DeleteDC(bufdc);
110     for(i=0;i<4;i++)
111         DeleteObject(girl[i]);
112     DeleteObject(bg);
113     ReleaseDC(hWnd,hdc);
114
115     PostQuitMessage(0);
116     break;
117 default:                                          //其他消息
118     return DefWindowProc(hWnd, message, wParam, lParam);
119 }
120 return 0;
121 }

```

### 程序说明

在消息处理函数中加入键盘消息处理，并按照使用者按下的【↑】、【↓】、【←】、【→】键来重设贴图坐标。

(1) 第8行：加入处理“WM\_KEYDOWN”消息。

(2) 第9行：判断按下按键的虚拟键码。

(3) 第11~13行：当按下按键为【Esc】键（VK\_ESCAPE）时结束程序。

(4) 第14~36行：按下【↑】键（VK\_UP）时的处理内容。先按照目前的移动方向来进行贴图坐标修正，并加入人物往上移动的量（每按下一次按键移动10个单位）来决定人物贴图坐标的x与y值，接着判断新坐标是否超出窗口区域，若有则再次进行修正。

(5) 第 37~104 行: 按照相同的方式处理按下【↓】键 (VK\_DOWN)、【←】键 (VK\_LEFT)、【→】键 (VK\_RIGHT) 时的操作。

程序代码: MyPaint()

```

1  //****自定义绘图函数*****
2  // 人物贴图坐标修正及窗口贴图
3  void MyPaint(HDC hdc)
4  {
5      int w,h;
6
7      SelectObject(bufdc,bg);
8      BitBlt(mdc,0,0,640,480,bufdc,0,0,SRCCOPY);
9
10     SelectObject(bufdc,girl[dir]);
11     switch(dir)
12     {
13         case 0:
14             w = 55;
15             h = 74;
16             break;
17         case 1:
18             w = 53;
19             h = 77;
20             break;
21         case 2:
22             w = 60;
23             h = 74;
24             break;
25         case 3:
26             w = 60;
27             h = 74;
28             break;
29     }
30     BitBlt(mdc,x,y,w,h,bufdc,num*w,h,SRCCAND);
31     BitBlt(mdc,x,y,w,h,bufdc,num*w,0,SRCPAINT);
32     BitBlt(hdc,0,0,640,480,mdc,0,0,SRCCOPY);
33
34     tPre = GetTickCount();          //记录此次绘图时间
35
36     num++;
37     if(num == 8)
38         num = 0;
39
40 }

```

程序说明

- (1) 第 7~8 行: 先在 mdc 中贴上背景图。
- (2) 第 10~29 行: 按照目前的移动方向取出对应人物的连续走动图, 并确定截取人物图的宽

度与高度。

(3) 第30~32行:按照目前的 $x$ 、 $y$ 值在 $mde$ 上进行透明贴图,然后显示在窗口画面上。

#### 运行结果

程序运行结果如图4-2所示。

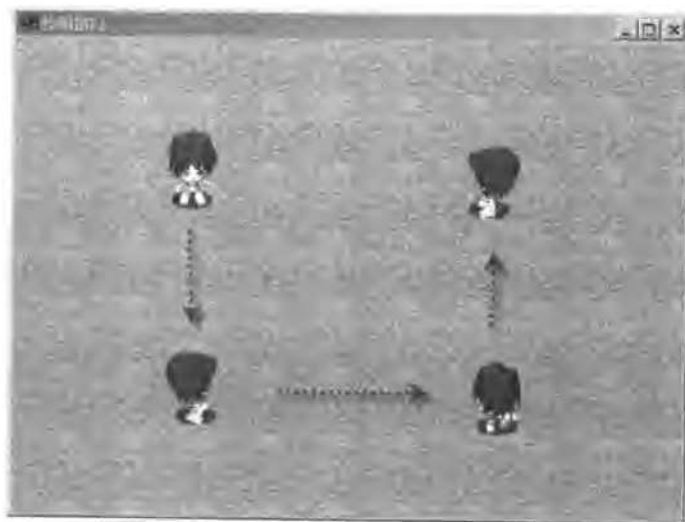


图4-2

通过在消息处理函数中取得按下按键虚拟键码的方式,可以很简单地对键盘输入操作进行处理。本书将 Windows 中所定义的虚拟键码整理并附在书后的附录中,供读者自行查阅。

## 4.2 鼠标输入消息

比起使用键盘作为游戏的输入装置,使用鼠标更为方便。这一节将介绍各种鼠标消息的处理方式及一些相关函数的应用方法。

与键盘相同,在 Windows 中定义了许多关于鼠标的输入消息。目前一般使用的鼠标通常为两键鼠标,而中间具有滚轮装置的鼠标也渐渐成了标准的规格。表4-3中列出了 Windows 中这类鼠标设备输入时的消息。

表4-3

消息代号	说明
WM_LBUTTONDOWNBLCLK	双击鼠标左键消息
WM_LBUTTONDOWN	单击鼠标左键消息
WM_LBUTTONUP	松开鼠标左键消息
WM_MBUTTONDOWNBLCLK	双击鼠标中键(滚轮)消息
WM_MBUTTONDOWN	单击鼠标中键(滚轮)消息
WM_MBUTTONUP	松开鼠标中键(滚轮)消息
WM_RBUTTONDOWNBLCLK	双击鼠标右键消息
WM_RBUTTONDOWN	单击鼠标右键消息



(续表)

消息代号	说 明
WM_RBUTTONDOWN	松开鼠标右键消息
WM_MOUSEMOVE	鼠标移动消息
WM_MOUSEWHEEL	鼠标滚轮转动消息

处理鼠标消息的方法跟处理键盘消息的方法类似，同样是在消息处理函数中加入要处理的鼠标消息类型。当鼠标消息发生时，输入的参数“wParam”与“lParam”则存储了鼠标状态的相关信息。下面对这两个参数加以说明。

## 1. lParam 参数

“lParam”参数的值可分为高位字节与低位字节两个部分，其中高位字节部分存储的是鼠标光标所在的 X 坐标值，低位字节部分存储的则是鼠标光标所在的 Y 坐标值。可以用下面的这两个函数来取得鼠标的坐标值：

```
WORD LOWORD(lParam 参数);    //返回鼠标光标所在的 X 坐标值
WORD HIWORD(lParam 参数);    //返回鼠标光标所在的 Y 坐标值
```

以上这两个函数所返回的鼠标光标位置的坐标是相对于内部窗口左上点坐标的。

## 2. wParam 参数

“wParam”参数的值记录着鼠标按键及键盘【Ctrl】键与【Shift】键的状态信息，通过表 4-4 中这些定义在“WINUSER.H”中的测试标志与“wParam”参数来检查上述按键的按下状态。

表 4-4

测试标志	说 明
MK_LBUTTON	按下鼠标左键
MK_MBUTTON	按下鼠标中（滚轮）键
MK_RBUTTON	按下鼠标右键
MK_SHIFT	按下【Shift】键
MK_CONTROL	按下【Ctrl】键

例如当某一鼠标消息发生时，要测试鼠标左键是否也被按下，程序代码如下。

```
if(wParam && MK_LBUTTON)
{
    //鼠标左键被按下
}
```

这是利用 wParam 参数与测试标志来测试鼠标键是否被按下的方法。当按键被按下时，条件式“wParam && MK\_LBUTTON”所传回的结果会是“true”。当然，若消息函数接收到“WM\_LBUTTONDOWN”消息，同样也可以知道鼠标键被按下而不必再去额外地做这样的测试。

再举个例子，如果要测试鼠标左键与【Shift】键的按下状态，那么程序代码如下。

```
if(wParam && MK_LBUTTON)
{
    if(wParam && MK_SHIFT)
```

```
{
    //单击鼠标左键
    //按下【Shift】键
}
else
{
    //单击鼠标左键
    //未按下【Shift】键
}
}
else
{
    if(wParam && MK_SHIFT)
    {
        //未单击鼠标左键
        //按下【Shift】键
    }
    else
    {
        //未单击鼠标左键
        //未按下【Shift】键
    }
}
```

通过这个例子应该可以清楚如何利用“wParam”参数与测试标志来测试鼠标键及【Shift】键和【Ctrl】键是否被按下的方法。

### 3. 滚轮消息

这里要特别一提的是鼠标滚轮转动消息(WM\_MOUSEWHEEL)。当鼠标滚轮转动消息发生时,“lParam”参数中的值同样是记录鼠标光标所在的坐标位置的,而“wParam”参数则分成高位字节与低位字节两部分,低位字节部分跟前面一样是存储鼠标键与【Shift】、【Ctrl】键的状态信息的,而高位字节部分的值会是“120”或“-120”。“120”表示鼠标滚轮向前转动,而“-120”则表示向后转动。

这里“wParam”高位组值与低位组值所用的函数同样是 HIWORD()与 LOWORD()。

```
HIWORD(wParam);    //高位组, 值为“120”或“-120”
LOWORD(wParam);    //低位组, 鼠标键及【Shift】和【Ctrl】键的状态信息
```

对各种鼠标输入消息及鼠标状态信息的获取方法有了基本的认识之后,下一节将介绍一些游戏程序中以鼠标来当做输入设备时经常会用到的相关函数。

## 4.3 鼠标相关函数

这一节将介绍一些 Windows API 中经常使用的鼠标的相关函数。通过这些函数的运用,可以更灵活地来控制鼠标的移动、显示及消息输入。

## 4.3.1 获取窗口外鼠标消息

这一小节先说明一下有关鼠标消息获取的概念。

前一小节列举了几种鼠标的输入消息，事实上这些消息只有鼠标光标位置在程序窗口中时才会起作用，否则程序就不会接收到鼠标的消息。

这样的状况似乎可以预见一些潜藏的危机，比如鼠标按键的按下与放开。举个例子，假如使用者单击了鼠标左键，窗口得到了“WM\_LBUTTONDOWN”消息，接着鼠标光标被移出窗口外而后放开。此时程序无法取得放开按键的消息，对程序而言，鼠标的左键依然是处于按下的状态（但事实上已经放开了），这样就有可能产生程序运行上的混乱。因此，为了确保程序可以正确地取得鼠标的输入消息，需要在必要的时候以下面的函数来设定窗口，以取得鼠标在窗口外所发出的消息。

```
HWND SetCapture( HWND hWnd ); //设定获取窗口外的鼠标消息
```

如果调用了上面的 SetCapture()函数，并输入要取得鼠标消息的窗口代号，那么便可取得鼠标在窗口外所发出的消息。这种方法也适用于多窗口的程序，与 SetCapture()函数相对应的函数为 ReleaseCapture()函数，用于释放窗口取得窗口外鼠标消息的函数。

```
BOOL ReleaseCapture(VOID); //释放获取窗口外的鼠标消息
```

## 4.3.2 设定鼠标光标位置

在游戏程序中，有时需要设定鼠标光标的位置。例如在射击游戏当中，玩家所操纵的飞机便是追踪鼠标光标来进行移动。程序一开始时必须将鼠标光标设定在要让飞机出现的位置上，不然若鼠标光标开始时在任意位置上，飞机也就会出现在任意的位置上，那么就不是希望出现的结果了。

这里介绍一个可以用来设定鼠标光标位置的函数。

```
BOOL SetCursorPos( int X坐标, int Y坐标 ); //设定鼠标光标位置
```

上面这个 SetCursorPos()函数中所设定的坐标是相对于屏幕左上角的屏幕坐标。事实上，以屏幕坐标的观点设定鼠标光标的位置会有一些小的问题。

在这里延续前面射击游戏的例子，假设要让飞机一开始出现在游戏窗口中(0,0)的位置上，那么必须以 SetCursorPos()函数来设定鼠标光标的坐标(0,0)。但是由于 SetCursorPos()函数输入的必须是屏幕坐标，而窗口坐标(0,0)对应到屏幕坐标上的值无法得知，因此需要用到 API 中的一个将窗口坐标转换到屏幕坐标的函数。

```
※BOOL ClientToScreen( HWND hWnd, //窗口坐标转换为屏幕坐标
                      LPPOINT 窗口点坐标 );
```

ClientToScreen()函数的第2个参数是一个 POINT 类型的点坐标。这个点坐标的值本来是窗口坐标，而调用运行这个函数后，其值会变成屏幕坐标。下面的一个程序片段将演示如何利用 ClientToScreen()函数与 SetCursorPos()函数将鼠标光标设定在窗口坐标(0,0)的位置上。

```
1 POINT pt; //声明点坐标
2
3 pt.x = 0; //设定x轴坐标值
4 pt.y = 0; //设定y轴坐标值
```

```

5
6 ClientToScreen(hwnd,&pt);    //将pt 的坐标值转换成屏幕坐标值
7 SetCursorPos(p.x,p.y);      //设定鼠标光标位置

```

在上面的程序中，第 3~4 行设定了“pt”的结构成员“pt.x”与“pt.y”分别为窗口 X 与 Y 轴上的坐标值（0.0）。调用 ClientToScreen()函数进行坐标转换之后，“pt.x”与“pt.y”变成窗口坐标（0.0）所对应的屏幕坐标值，最后以屏幕坐标（p.x,p.y）设定鼠标光标的位置。

以上就是关于坐标转换的概念及设定鼠标光标位置的方法。此外，API 中还有另一个将屏幕坐标转换为窗口坐标的函数。如下所示：

```

BOOL ScreenToClient( HWND hwnd,                //屏幕坐标转换为窗口坐标
                    LPPOINT 屏幕点坐标 );

```

### 4.3.3 显示与隐藏鼠标光标

在游戏程序中经常需要隐藏或者显示鼠标光标，这项工作只需要利用下面的这个函数即可完成。

```

int ShowCursor(BOOL true 或 false);    //隐藏及显示鼠标光标

```

当输入的参数为“true”时，表示显示鼠标光标；输入“false”则是隐藏光标。这里所谓的隐藏及显示指的是鼠标光标的图案，事实上不论隐藏或者显示鼠标光标还是存在的。

### 4.3.4 限制鼠标光标移动区域

API 中提供的 ClipCursor()函数可以用来设置限制鼠标光标的移动区域和解除鼠标光标移动区域的限制。

```

BOOL ClipCursor(CONST RECT 移动区域矩形);    //限制鼠标光标移动区域
BOOL ClipCursor(NULL);                        //解除限制

```

要设定鼠标光标移动区域时，输入这个函数中的参数是一个 RECT 类型的矩形结构。该结构的内容如下。

```

typedef struct RECT {
    LONG left;        //矩形区域右上点 x 坐标
    LONG top;         //矩形区域右上点 y 坐标
    LONG right;       //矩形区域左上点 x 坐标
    LONG bottom;      //矩形区域左下点 y 坐标
} RECT;

```

在什么情况下需要限制鼠标光标移动区域？还是以前面射击游戏为例子，在射击游戏中，飞机移动的区域是在窗口中，而且飞机追踪鼠标光标来移动。如果程序没有限制鼠标光标的移动区域，一旦鼠标光标移到窗口外，那么飞机就不知该飞向哪里。

因此，在这样的情况下，通常需要限制鼠标光标，使其仅能在游戏窗口中移动。下面列出了两个可用来取得窗口外部区域及内部区域的 API 函数。

```

BOOL GetWindowRect( HWND hwnd,LPRECT 矩形结构 );    //取得窗口外部区域矩形

```

```
BOOL GetClientRect( HWND hWnd,LPRECT 矩形结构); //取得窗口内部区域矩形
```

这两个函数的第2个参数都是输入一个矩形结构，用来存储函数所取得的外部区域或者内部区域。由图4-3可以看出窗口外部区域与内部区域的不同之处。



图 4-3

这里要特别注意的是，若用 `GetWindowRect()` 函数取得窗口外部区域，那么所得到的矩形区域的左上及右下的坐标点是属于屏幕坐标；若以 `GetClientRect()` 函数取得窗口内部区域，那么得到的矩形区域的左上及右下坐标点则是窗口坐标。一般情况下希望鼠标光标只能在内部区域中移动，这样会比限制在外部区域中移动更精确，不过由于限制鼠标光标移动区域的 `ClipCursor()` 函数中输入的矩形区域必须是屏幕坐标，因此如果取得的是窗口内部区域，那么还必须进行将窗口坐标转换成屏幕坐标的操作。下面以程序代码来说明将鼠标光标限制在窗口内部区域移动的过程。

```
1  RECT rect;
2  POINT lt,rb;
3
4  GetClientRect(hWnd,&rect); //取得窗口内部矩形
5
6  //将矩形左上点坐标存入lt中
7  lt.x = rect.left;
8  lt.y = rect.top;
9
10 //将矩形右下点坐标存入rb中
11 rb.x = rect.right;
12 rb.y = rect.bottom;
13
14 //将lt和rb的窗口坐标转换为屏幕坐标
15 ClientToScreen(hWnd,&lt);
16 ClientToScreen(hWnd,&rb);
17
18 //以屏幕坐标重新设定矩形区域
19 rect.left = lt.x;
20 rect.top = lt.y;
21 rect.right = rb.x;
22 rect.bottom = rb.y;
23
24 //限制鼠标光标移动区域
25 ClipCursor(&rect);
```

上面的程序代码中加入了注释，应该可以了解到取得窗口内部区域、窗口坐标转换成屏幕坐标、

重新设定矩形区域,最后以屏幕坐标的矩形区域来限制鼠标光标移动区域的整个步骤。

这一节一口气介绍了不少跟鼠标相关的 Windows API 函数,并以射击游戏为例说明了这些函数在游戏中使用的时机。接下来就以一个简单的飞机射击范例来示范鼠标消息的处理方法,以及这一节里所介绍过的一些鼠标相关函数的使用方法。

## » 范例 ch4\_2

处理鼠标移动消息使飞机在窗口中移动;处理单击鼠标左键消息来让飞机发射子弹;示范设定鼠标光标位置、隐藏鼠标光标与限制鼠标光标移动区域的方法。

### 程序代码: 结构定义与全局变量声明

```
1 //定义结构
2 struct BULLET
3 {
4     int x,y;
5     bool exist;
6 };
7
8 //全局变量声明
9 HINSTANCE hInst;
10 HBITMAP bg,ship,bullet;
11 HDC      hdc,mdc,bufdc;
12 HWND hWnd;
13 DWORD    tPre,tNow;
14 int       x,y,nowX,nowY;
15 int       w=0,bcount;
16 BULLET b[30];
```

### 程序说明

(1) 第2~6行: 定义代表飞机子弹的结构“bullet”。结构成员“x”和“y”用来记录子弹的坐标,“exist”代表子弹是否存在。

(2) 第10行: 声明3个位图对象“bg”、“ship”和“bullet”,分别用来存储背景图、飞机图和子弹图。

(3) 第14行: 变量“x”和“y”为鼠标光标所在坐标,变量“nowX”与“nowY”代表飞机的坐标,也是贴图坐标。

(4) 第15行: 变量“w”为滚动背景所要裁切的区域宽度,变量“bcount”记录飞机现有的子弹数目。

(5) 第16行: 声明一个“bullet”类型的数组,用来存储飞机发出的子弹。

### 程序代码: InitInstance()

```
1 //****初始化函数*****
2 // 1.设定飞机初始位置
3 // 2.设定鼠标光标位置及隐藏
4 // 3.限制鼠标光标移动区域
5 BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
6 {
```

```

7  HBITMAP bmp;
8  POINT pt,lt,rb;
9  RECT rect;
10
11 hInst = hInstance;
12
13 hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
14     CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
15
16 if (!hWnd)
17 {
18     return FALSE;
19 }
20
21 MoveWindow(hWnd,10,10,640,480,true);
22 ShowWindow(hWnd, nCmdShow);
23 UpdateWindow(hWnd);
24
25 hdc = GetDC(hWnd);
26 mdc = CreateCompatibleDC(hdc);
27 bufdc = CreateCompatibleDC(hdc);
28
29 bmp = CreateCompatibleBitmap(hdc,640,480);
30 SelectObject(mdc,bmp);
31
32 bg = (HBITMAP)LoadImage(NULL,"bg.bmp",IMAGE_BITMAP,648,480,LR_LOADFROMFILE);
33 ship = (HBITMAP)LoadImage(NULL,"ship.bmp",IMAGE_BITMAP,100,148,
    LR_LOADFROMFILE);
34 bullet = (HBITMAP)LoadImage(NULL,"bullet.bmp",IMAGE_BITMAP,10,20,
    LR_LOADFROMFILE);
35
36 x = 300;
37 y = 300;
38 nowX = 300;
39 nowY = 300;
40
41 //设定鼠标光标位置
42 pt.x = 300;
43 pt.y = 300;
44 ClientToScreen(hWnd,&pt);
45 SetCursorPos(pt.x,pt.y);
46
47 ShowCursor(false); //隐藏鼠标光标
48
49 //限制鼠标光标移动区域
50 GetClientRect(hWnd,&rect);
51 lt.x = rect.left;
52 lt.y = rect.top;
53 rb.x = rect.right;

```

```

54  rb.y = rect.bottom;
55  ClientToScreen(hWnd,&lt);
56  ClientToScreen(hWnd,&rb);
57  rect.left = lt.x;
58  rect.top = lt.y;
59  rect.right = rb.x;
60  rect.bottom = rb.y;
61  ClipCursor(&rect);
62
63  MyPaint(hdc);
64
65  return TRUE;
66 }

```

**程序说明**

- (1) 第 29~30 行: 建立空位图并置入 hdc 中。
- (2) 第 36~39 行: 设定代表鼠标光标坐标的“x”和“y”值, 并设定飞机贴图坐标的“nowX”和“nowY”的值为预设初始坐标(300,300)。
- (3) 第 42~45 行: 将实际的鼠标光标移到(300,300)的位置上。
- (4) 第 47 行: 隐藏鼠标光标。
- (5) 第 50~61 行: 取得窗口内部区域并进行屏幕坐标转换, 限制鼠标光标移动区域为窗口内部。

**程序代码: WinProc**

```

1  //****消息处理函数*****
2  // 1.处理 WM_LBUTTONDOWN 消息发射子弹
3  // 2.处理 WM_MOUSEMOVE 消息设定飞机贴图坐标
4  // 3.在窗口结束时恢复鼠标移动区域
5  LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
    lParam)
6  {
7      int i;
8
9      switch (message)
10     {
11         case WM_KEYDOWN: //按键按下消息
12             if(wParam==VK_ESCAPE) //按下【Esc】键
13                 PostQuitMessage(0);
14             break;
15         case WM_LBUTTONDOWN: //单击鼠标左键消息
16             for(i=0;i<30;i++)
17             {
18                 if(!b[i].exist)
19                 {
20                     b[i].x = nowX; //子弹 x 坐标
21                     b[i].y = nowY + 30; //子弹 y 坐标
22                     b[i].exist = true;
23                     bcount++; //累加子弹数目

```



```

24         break;
25     }
26 }
27 case WM_MOUSEMOVE:
28     x = LOWORD(lParam);           //取得鼠标 x 坐标
29     if(x > 530)
30         x = 530;
31     else if(x < 0)
32         x = 0;
33
34     y = HIWORD(lParam);           //取得鼠标 y 坐标
35     if(y > 380)
36         y = 380;
37     else if(y < 0)
38         y = 0;
39
40     break;
41 case WM_DESTROY:                   //窗口结束消息
42     ClipCursor(NULL);              //恢复鼠标移动区域
43
44     DeleteDC(mdc);
45     DeleteDC(bufdc);
46     DeleteObject(bg);
47     DeleteObject(bullet);
48     DeleteObject(ship);
49     ReleaseDC(hWnd, hdc);
50
51     PostQuitMessage(0);
52     break;
53 default:                            //其他消息
54     return DefWindowProc(hWnd, message, wParam, lParam);
55 }
56 return 0;
57 }

```

## 程序说明

在消息处理函数中，在鼠标移动时设定飞机移动的目的点，并当单击鼠标左键时产生子弹。

(1) 第 11~14 行：处理按键按下消息，当按下【Esc】键时则结束程序。在这个范例中，由于设定鼠标移动区域仅在窗口内部中，因此无法单击窗口上方的关闭按钮来关闭程序，只能按下【Esc】键来结束程序。

(2) 第 16~26 行：当单击鼠标左键时，为了弹数组“b[]”中不存在子弹的元素（b[i].exist 为 false 的元素）加入一颗子弹，并按照目前飞机的位置“nowX”与“nowY”来设定子弹的坐标，将结构成员“exist”设为“true”代表子弹已产生。当子弹产生后，累加代表了弹数目的变量“bcount”的值。

(3) 第 28~38 行：当鼠标移动时，取得鼠标光标的坐标值，并存到变量“x”与“y”中。坐标（x,y）代表飞机移动的目的点。在这个程序中，飞机移动的轨迹是向着目的点慢慢移动，这样比较符合真实的飞行状况。

(4) 第 29~32、35~38 行: 设定实际贴图的临界坐标, 使得飞机贴图时不会没入窗口中。

(5) 第 42 行: 当程序结束时, 调用 ClipCursor(NULL)函数恢复鼠标光标的移动范围。

**程序代码: MyPaint()**

```
1  //****自定义绘图函数*****
2  // 1.设定飞机坐标并进行贴图
3  // 2.设定所有子弹坐标并进行贴图
4  // 3.显示真正的鼠标光标所在坐标
5  void MyPaint(HDC hdc)
6  {
7      char str[20] = "";
8      int i;
9
10     //贴上背景图
11     SelectObject(bufdc,bg);
12     BitBlt(mdc,0,0,w,480,bufdc,640-w,0,SRCCOPY);
13     BitBlt(mdc,w,0,640-w,480,bufdc,0,0,SRCCOPY);
14
15     if(nowX < x)
16     {
17         nowX += 10;
18         if(nowX > x)
19             nowX = x;
20     }
21     else
22     {
23         nowX -=10;
24         if(nowX < x)
25             nowX = x;
26     }
27
28     if(nowY < y)
29     {
30         nowY += 10;
31         if(nowY > y)
32             nowY = y;
33     }
34     else
35     {
36         nowY -= 10;
37         if(nowY < y)
38             nowY = y;
39     }
40
41     //贴上飞机图
42     SelectObject(bufdc,ship);
43     BitBlt(mdc,nowX,nowY,100,74,bufdc,0,74,SRCCAND);
44     BitBlt(mdc,nowX,nowY,100,74,bufdc,0,0,SRCPAINT);
45
```

```

46 SelectObject(bufdc,bullet);
47 if(bcount!=0)
48     for(i=0;i<30;i++)
49         if(b[i].exist)
50             {
51                 //贴上子弹图
52                 BitBlt(mdc,b[i].x,b[i].y,10,10,bufdc,0,10,SRCCAND);
53                 BitBlt(mdc,b[i].x,b[i].y,10,10,bufdc,0,0,SRCPAINT);
54
55                 b[i].x -= 10;
56                 if(b[i].x < 0)
57                     {
58                         bcount--;
59                         b[i].exist = false;
60                     }
61             }
62
63 //显示鼠标坐标
64 sprintf(str,"X坐标: %d  ",x);
65 TextOut(mdc,0,0,str,strlen(str));
66 sprintf(str,"Y坐标: %d  ",y);
67 TextOut(mdc,0,20,str,strlen(str));
68
69 BitBlt(hdc,0,0,640,480,mdc,0,0,SRCCOPY);
70
71 tPre = GetTickCount();
72
73 w += 10;
74 if(w==640)
75     w = 0;
76 }

```

## 程序说明

MyPaint()函数将按照顺序进行滚动背景、飞机的贴图和子弹的贴图，并输出鼠标光标的坐标消息，然后在窗口中显示最后的结果画面。

(1) 第 15~39 行：计算飞机的贴图坐标，设定每次进行飞机贴图时，其贴图坐标(nowX,nowY)会以 10 个单位慢慢向鼠标光标所在的目的点(x,y)接近，直到两坐标相同为止。

(2) 第 46~53 行：完成飞机贴图之后进行子弹的贴图，先判断子弹数目“bcount”的值是否为“0”。若不为“0”，则对子弹数组中各个还存在的子弹按照其所在坐标(b[i].x,b[i].y)循环进行贴图的操作。

(3) 第 55~60 行：设定下次子弹的坐标。在这个范例中，子弹是由右向左发射前进的，因此，每次其 X 轴上的坐标值递减 10 个单位，这样下次贴图时便会产生往左移动的效果。而如果子弹下次的坐标已超出窗口的可见范围(b[i].x<0)，那么将子弹设为不存在，并将子弹总数“bcount”变量的值减“1”。

(4) 第 64~67 行：输出目前鼠标光标所在的坐标值，也就是飞机移动的目的点。

**运行结果**

程序运行结果如图 4-4 所示。

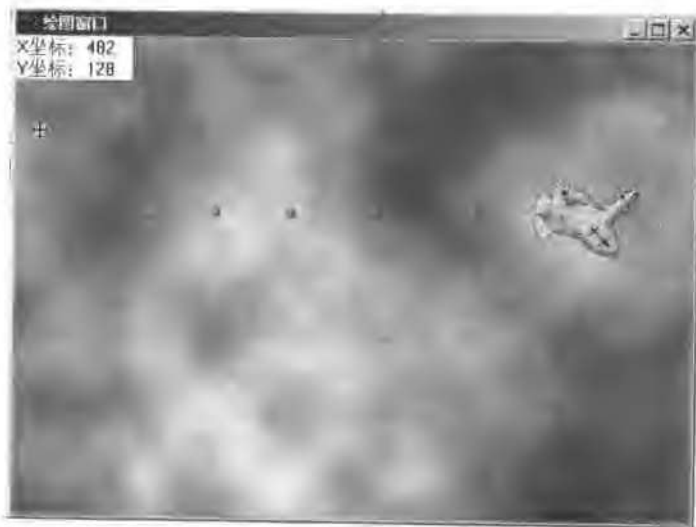


图 4-4

在这一章中介绍了如何处理通过键盘或者鼠标的输入消息，也完成了一个效果不错的飞机射击程序。在后面第 10 章中还会进一步地使用 DirectX 中的 DirectInput 函数来处理各种输入装置（包含键盘、鼠标与游戏杆）的输入消息，制作高级的输入控制功能。

## 课后重点整理

- Windows 系统对所有键盘的按键定义出一组通用的“虚拟键码”，也就是说在 Windows 系统下，所有的按键都会被视为虚拟键（包含鼠标键在内），而每一个虚拟键都有其对应的一个虚拟键码。
- Windows 系统是一个消息驱动的环境，一旦使用者在键盘上进行了输入操作，那么系统便会接收到对应的键盘消息。
- Windows 系统本身定义了一组“系统键”，这些按键通常是【Alt】键与其他按键的组合。系统键对于 Windows 系统本身有一些特定的作用。
- 当鼠标消息发生时，输入的参数“wParam”与“lParam”存储了鼠标状态的相关信息。其中参数“lParam”的值可分为高位字组与低位字组两个部分，高位字组部分存储的是鼠标光标所在的 X 坐标值，低位字组部分存储的则是鼠标光标所在的 Y 坐标值；参数“wParam”的值记录的是鼠标按键及键盘【Ctrl】键与【Shift】键的状态信息。
- API 中提供的 ClipCursor() 函数可以用来设置限制鼠标光标的移动区域及解除鼠标光标移动区域的限制。

### 课后练习

1. 请问最常见的 3 种键盘消息是什么？并试着说明其代表的意义。

2. 观察消息处理函数中输入的两个参数“wParam”与“lParam”:

```
LRESULT CALLBACK WndProc(HWND hWnd,
                           UINT message,
                           WPARAM wParam,
                           LPARAM lParam)
```

请试着解释这两个参数的意义。

3. 目前鼠标通常为两键鼠标或中间具有滚动装置的鼠标。表 4-5 中列出了 Windows 中这类鼠标设备输入时的消息, 请说明各消息代号所代表的意义。

表 4-5

消息代号	说 明
WM_LBUTTONDOWN	
WM_LBUTTONDOWN	
WM_LBUTTONUP	
WM_MBUTTONDOWN	
WM_MBUTTONDOWN	
WM_MBUTTONUP	
WM_RBUTTONDOWN	
WM_RBUTTONDOWN	
WM_RBUTTONUP	
WM_MOUSEMOVE	
WM_MOUSEWHEEL	

4. 如果要测试鼠标左键与【Shift】键的按下状态, 那么程序的判断方法是什么? 请以程序语法简略说明。

# 第 5 章 游戏人工智能

## 5.1 移动型游戏 AI

我们常听到的 AI (Artificial Intelligence) 即人工智能, 是一门内容相当广的研究领域。它的主要目的是要让计算机本身按照某些法则来模拟出类似人类般的思考力与预测能力, 并结合计算机具有快速数学运算能力的优点, 创造出计算机在各方面的有效应用。

这一章要讨论的游戏人工智能, 实际上只是整个人工智能研究领域中的一小部分。这里用不到像类神经网络、基因算法、模糊逻辑等复杂的人工智能理论。相反, 只需利用自己本身的思考模式去赋予游戏中角色的判断能力, 来进行某些特定的行为, 这样便可拓展出属于游戏自己本身的人工智能, 而这也是一般游戏开发过程中最常见的方式。

接下来将探讨一些游戏 AI 的基本概念, 包括游戏角色的移动、路径搜寻和计算机的决策方式等。通过这些主题的讨论, 可增加对游戏 AI 设计上的基本认识, 激发设计游戏 AI 的灵感。

### 5.1.1 追逐移动

凡是在游戏中会移动的物体, 实际上几乎都涉及移动型的游戏 AI, 例如游戏中怪物追逐或者躲避玩家和计算机角色的移动等都是移动型 AI 的例子。这一小节中先来介绍在游戏程序中经常会看到的怪物追逐玩家这种追逐移动的设计方式。

追逐移动是通过计算机控制角色朝某一目标物接近来实现, 要设计出这样的物体移动模式很简单。以怪物追逐玩家的例子来说, 只要在每次进行窗口贴图时, 将怪物的所在坐标与玩家角色的所在坐标做比较, 递增或递减怪物 X、Y 轴上的贴图坐标, 使得怪物每次进行贴图时渐渐朝玩家角色所在的位置接近, 便可产生追逐移动的效果。下面便是一个典型的怪物追逐玩家的移动 AI 算法, 其中“怪物 X”、“怪物 Y”、“玩家 X”、“玩家 Y”分别用来表示怪物及玩家在 X 与 Y 轴上的贴图坐标。

```
if (怪物 X > 玩家 X)
    怪物 X--;
else
    怪物 X++;

if (怪物 Y > 玩家 Y)
    怪物 Y--;
else
    怪物 Y++;
```

以前面所说明的概念来看这段算法的内容应该不难理解, 是让怪物能正确地往玩家角色所在的目的地移动。不过一般在游戏程序当中, 常会按照各种不同的情况 (例如怪物本身的追逐能力、游戏等级的难易度等) 来加入怪物追逐移动的不确定性, 以提高计算机角色移动的多样化。

下面来看一个怪物追逐玩家的算法例子。这段算法是以上面的算法为基础修改，使得进行追逐移动的怪物会按照自身生命值的多寡来决定是否进行追逐。在每次计算下次的位置坐标时，也只有 2/3 的几率能正确地朝向玩家。其中以“怪物 HP”来表示怪物当前的生命值。

```
if(怪物HP>200)           //生命值大于 200 时才追
{
    p = rand()%3;         //取随机数除以 3 的余数
    if(p!=1)              //余数不为 1 时进行追逐
    {
        if(怪物X>玩家X)
            怪物X--;
        else
            怪物X++;

        if(怪物Y>玩家Y)
            怪物Y--;
        else
            怪物Y++;
    }
    else
        怪物HP+=5;        //怪物不动，休息补血
```

上面的这段怪物追逐的算法看起来比较具有弹性且符合真实的情况。在了解设计计算机角色追逐移动的方式之后，下面就来看一个完整的追逐移动范例程序。

## 》》范例 ch5\_1

在范例 ch4\_2（飞机在循环背景上移动）中，加入 3 只追逐飞机移动的小鸟。

### 程序代码：全局变量声明

```
1  //全局变量声明
2  HINSTANCE  hInst;
3  HBITMAP    bg, ship, bird;
4  HDC        hdc, mdc, bufdc;
5  HWND       hWnd;
6  DWORD      tPre, tNow;
7  int        x, y, nowX, nowY;
8  int        w=0;
9  POINT      p[3];
```

### 程序说明

第 9 行：声明“POINT”类型的点坐标数组“p[]”，用来记录窗口中 3 只小鸟的贴图坐标。在 InitInstance()函数中设定数组中各个元素的初值。范例中使用了如图 5-1 所示的这张 122×122 的小鸟位图。

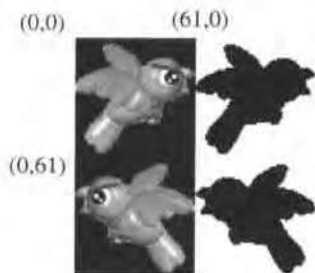


图 5-1

当目标飞机在小鸟的右边时，以图中上面的小鸟图案进行贴图；当目标飞机在小鸟的左边时，则以图中下面小鸟的图案进行贴图。

程序代码：MyPaint()

```

1  //****自定义绘图函数*****
2  // 1.设定飞机坐标并进行贴图
3  // 2.设定小鸟坐标并进行贴图
4  void MyPaint (HDC hdc)
5  {
6      int i;
7
8      //贴上背景图
9      SelectObject (bufdc,bg);
10     BitBlt (mdc,0,0,w,480,bufdc,640-w,0,SRCCOPY);
11     BitBlt (mdc,w,0,640-w,480,bufdc,0,0,SRCCOPY);
12
13     //贴上飞机图
14     if (nowX < x)
15     {
16         nowX += 10;
17         if (nowX > x)
18             nowX = x;
19     }
20     else
21     {
22         nowX -= 10;
23         if (nowX < x)
24             nowX = x;
25     }
26
27     if (nowY < y)
28     {
29         nowY += 10;
30         if (nowY > y)
31             nowY = y;
32     }
33     else
34     {

```



```

35     nowY -= 10;
36     if(nowY < y)
37         nowY = y;
38 }
39 SelectObject(bufdc, ship);
40 BitBlt(mdc, nowX, nowY, 100, 74, bufdc, 0, 74, SRCAND);
41 BitBlt(mdc, nowX, nowY, 100, 74, bufdc, 0, 0, SRCPAINT);
42
43 //贴上小鸟图
44 SelectObject(bufdc, bird);
45 for(i=0; i<3; i++)
46 {
47     if(rand()%3 != 1)                //设定 2/3 几率进行追逐
48     {
49         if(p[i].y > nowY-16)
50             p[i].y -= 5;
51         else
52             p[i].y += 5;
53
54         if(p[i].x > nowX-25)
55             p[i].x -= 5;
56         else
57             p[i].x += 5;
58     }
59
60     if(p[i].x > nowX-25)              //判断小鸟移动方向
61     {
62         BitBlt(mdc, p[i].x, p[i].y, 61, 61, bufdc, 61, 61, SRCAND);
63         BitBlt(mdc, p[i].x, p[i].y, 61, 61, bufdc, 0, 61, SRCPAINT);
64     }
65     else
66     {
67         BitBlt(mdc, p[i].x, p[i].y, 61, 61, bufdc, 61, 0, SRCAND);
68         BitBlt(mdc, p[i].x, p[i].y, 61, 61, bufdc, 0, 0, SRCPAINT);
69     }
70 }
71
72 BitBlt(hdc, 0, 0, 640, 480, mdc, 0, 0, SRCCOPY);
73
74 tPre = GetTickCount();
75
76 w += 10;
77 if(w==640)
78     w = 0;
79 }
  
```

## 程序说明

(1) 第 14~41 行: 按照实际鼠标光标的位置 (x,y) 设定飞机的贴图坐标 (nowX,nowY) 并进行飞机的贴图。

(2) 第 44~70 行: 进行小鸟追逐移动坐标计算及贴图操作。其中第 47 行程序代码设定小鸟有  $\frac{2}{3}$  的几率进行追逐移动。第 48~58 行设定移动的贴图坐标。第 60~69 行以目前飞机所在坐标点 “nowX” 减掉 25 个单位为中心点来判断贴上小鸟向右或向左飞行的图案。

#### 运行结果

程序运行结果如图 5-2 所示。

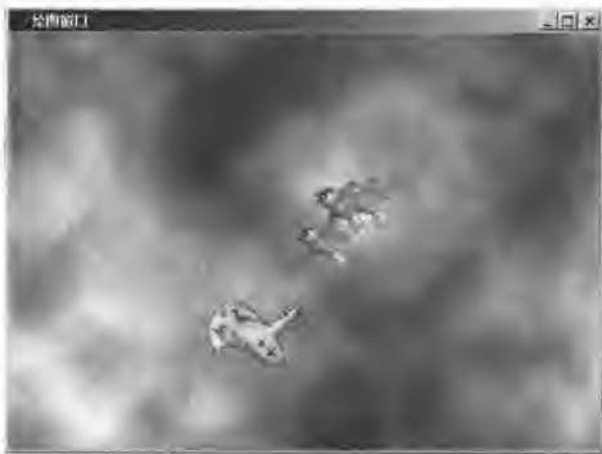


图 5-2

小鸟在窗口中追逐飞机移动, 按下【Esc】键结束程序的运行。

#### VC++技巧 人工智能理论

- ◆ 类神经网络: 以多个联结处理器负责不同单元的处理, 仿真人类大脑思考与学习能力的人工智能理论。
- ◆ 基因算法: 利用仿真自然界适者生存的进化原理, 对于问题产生最佳解决方案的人工智能理论。
- ◆ 模糊逻辑: 以一种判断推理 (if-else) 的方式来产生最佳猜测的决定, 有别于一般以数学运算为基础的人工智能理论。

### 5.1.2 躲避移动

从前面的追逐移动大概可以推知躲避移动是怎么一回事。与追逐移动朝着目标前进的目的刚好相反, 躲避移动的目的是远离目标。下面就以计算机怪物躲避玩家的例子来看看躲避移动基本的算法。

```
if(怪物X>玩家X)
    怪物X++;
else
    怪物X--;

if(怪物Y>玩家Y)
    怪物Y++;
else
```

怪物Y--;

上面的这段算法，其中的判断式与追逐移动是相同的，不过每次重设怪物的贴图坐标时，则会越来越远离玩家角色的所在位置。

## 5.1.3 模式移动

由于计算机角色的移动通常并不完全只进行像追逐或者躲避这样的单一的移动模式，因此程序中经常替计算机角色定义出一组移动的模式，其中可能包含多种基本的移动模式，例如追逐、躲避、随机、固定移动。计算机角色会随着游戏情节的改变按照定义的不同移动模式进行移动，这就是模式移动。

这里以计算机怪物视玩家的强弱进行移动的例子来说明模式移动的方法。假设游戏中定义的怪物移动模式包含了追逐、躲避与随机移动3种模式，那么怪物进行这些移动的时机如图5-3所示。

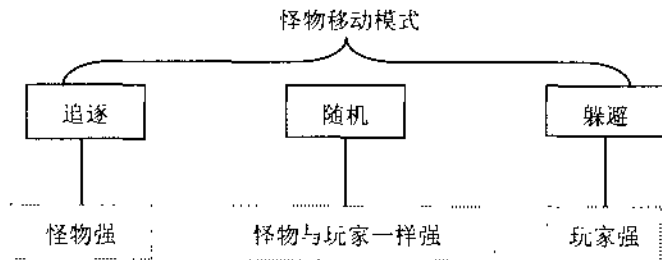


图 5-3

模式移动的设计可以让计算机角色拥有更高级且富变化性的移动行为。像上面这个例子中，怪物通过比较玩家与自身的强弱来决定进行何种方式的移动，其中涉及怪物本身的思考及思考后的行为动作，因此这部分与下一节中要探讨的行为型 AI 相关。

## 5.2 行为型游戏 AI

行为型游戏 AI 主要是通过计算机角色本身的判断思考，然后产生对应行为的 AI。在设计行为型的游戏 AI 时，通常会利用到一连串的“if-else”判断、数学运算，或者一些数据结构的概念。在这一节中，将讨论如何来构思并赋予游戏计算机角色的行为 AI。

### 5.2.1 计算机角色的思考与行为

首先来谈谈游戏程序中计算机角色的思考与行为，这部分内容事实上与平日对于各种不同事件情况进行分析思考，进而加以处理的概念是一样的。至于在游戏程序中如何让计算机角色拥有判断状况的思考能力，并按照判断后的结果进行相应的行为动作？可以利用“if-else”和“switch-case”这类的判断式来完成游戏中这类行为型 AI 的设计。

这里举个 RPG 游戏中怪物行为的例子进行说明。假设某一种怪物在对战时具有下面的几种行为：

(1) 普通攻击；

- (2) 施放攻击魔法;
- (3) 使尽全力攻击;
- (4) 补血;
- (5) 逃跑。

那么根据以上的几种怪物的行为,可编写出如下的一段算法,用来仿真怪物在对战时的行为模式。

```
1  if(生命值>20)                //生命值大于 20
2  {
3      if(rand()%10 != 1)        //进行普通攻击的几率为 9/10
4          普通攻击;
5      else
6          施放攻击魔法;
7  }
8  else                          //生命值小于 20
9  {
10     switch(rand()%5)
11     {
12         case 0:
13             普通攻击;
14             break;
15         case 1:
16             施放攻击魔法;
17             break;
18         case 2:
19             使尽全力攻击;
20             break;
21         case 3:
22             补血;
23             break;
24         case 4:
25             逃跑;
26             if(rand()%3 == 1)    //逃跑成功几率为 1/3
27                 逃跑成功;
28             else
29                 逃跑失败;
30             break;
31     }
32 }
```

在上面的这段算法中,利用“if-else”判断式判断怪物的生命值是否大于“20”。当怪物生命值大于“20”时,怪物有 9/10 的几率进行普通攻击及 1/10 的几率进行施放攻击魔法;假设当怪物受到了严重伤害且生命值小于“20”时,第 10 行程序代码通过“switch”叙述判断“rand()%5”的结果来进行对应的行为,因而怪物有可能会进行普通攻击、施放攻击魔法、使尽全力攻击、补血和逃跑等动作,而这些怪物行为的发生几率各为 1/5,其中第 26 行则是设定怪物逃跑成功的几率为 1/3。

事实上像上面这样利用“if-else”、“switch”叙述,使计算机角色进行状况判断,并产生对应的行为动作就是行为型游戏 AI 设计的基本精神。下面来看一个简单的 RPG 玩家与怪物对战的范例程

序。这个范例采用玩家与计算机轮流攻击的模式，其中怪物部分的行为 AI 将按照上面的算法以实际的程序代码加以实作，玩家部分则主要是下达攻击命令，两者之间的对战状态以文字消息来显示。运行时的画面如图 5-4 所示。



图 5-4

## » 范例 ch5\_2

在怪物与玩家对战模式中，设计计算机控制怪物的行为型 AI。

程序代码：结构定义与全局变量声明

```

1  //定义结构
2  struct chr
3  {
4      int      nHp;      //目前生命值
5      int      fHp;      //最大生命值
6      int      lv;       //等级
7      int      w;        //加权值
8      int      kind;     //怪物的行为代号
9  };
10
11 //全局变量声明
12 HINSTANCE hInst;
13 HBITMAP   bg,sheep,girl,skill,slash,magic,recover,game;
14 HDC       hdc,mdc,bufdc;
15 HWND      hwnd;
16 DWORD     tPre,tNow;
17 int        pNum,f,txtNum;
18 bool       attack,over;
19 chr        player,monster;
20 char       text[5][100];
    
```

### 程序说明

(1) 第 2~9 行：定义“chr”结构。各个结构成员用来存储怪物或者玩家的能力及状态信息，

其中的“w”成员是一个加权值,用来计算怪物或玩家攻击时的伤害力。

(2) 第 13 行: 声明程序中所用到的位图对象,其显示的图案说明如表 5-1 所示。

表 5-1

对象名称	说 明
bg	背景图
sheep	怪物图
girl	玩家图
skill	攻击命令图
slash	普通攻击图
magic	怪物魔法攻击图
recover	怪物恢复魔法图
game	游戏结束图

(3) 第 17 行: 声明“pNum”为玩家人物跑动的图号变量;“f”变量代表每一回合攻击时更新的画面数,利用这个变量的调整,可以设定每次攻击画面(普通攻击、魔法攻击、魔法补血等)效果暂留的时间长短,并在固定的时刻进行玩家与怪物生命值计算及一些事件的处理;“txtNum”变量是用来记录目前所要显示消息的总数,在这个范例中设定显示消息的上限数目为“5”。

(4) 第 18 行: 声明两个布尔变量“attack”与“over”。“attack”变量用来判断玩家是否按下攻击命令的图标,一旦玩家下达攻击命令,便开始一回合玩家与怪物的对战;“over”变量则记录游戏是否结束,当玩家或者怪物有一方的生命值降到“0”以下,“over”的值会设为“true”。

(5) 第 19 行: 由前面所建立的 chr 结构声明两个变量“player”与“monster”分别代表玩家与怪物。

(6) 第 20 行: 声明一个二维的字符缓冲区,用来存储要显示的对战消息。

#### 程序代码: 函数声明

```

1  //函数声明
2  ATOM          MyRegisterClass(HINSTANCE hInstance);
3  BOOL          InitInstance(HINSTANCE, int);
4  LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
5  void          MyPaint(HDC hdc);
6  void          MsgInsert(char*);
7  void          CheckDie(int hp,bool player);

```

#### 程序说明

(1) 第 6 行: 声明一个函数 MsgInsert(), 此函数用来新增要显示的消息到“text”缓冲区中。当显示消息的数目达到显示的上限数目时,则删除最先的消息。

(2) 第 7 行: 声明一个函数 CheckDie(), 在每次玩家或怪物进行攻击后会调用这个函数来判断被攻击的一方的生命值是否降至“0”以下。若生命值降至“0”以下则将“over”变量设为“true”,显示游戏结束画面。

#### 程序代码: InitInstance()

```

1  /***初始函数*****
2  // 初始玩家及怪物的各项状态
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {

```

```

5  HBITMAP bmp;
6  hInst = hInstance;
7
8  hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
9      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
10
11  if (!hWnd)
12  {
13      return FALSE;
14  }
15
16  MoveWindow(hWnd, 10, 10, 640, 480, true);
17  ShowWindow(hWnd, nCmdShow);
18  UpdateWindow(hWnd);
19
20  hdc = GetDC(hWnd);
21  mdc = CreateCompatibleDC(hdc);
22  bufdc = CreateCompatibleDC(hdc);
23
24  bmp = CreateCompatibleBitmap(hdc, 640, 480);
25  SelectObject(mdc, bmp);
26
27  bg = (HBITMAP)LoadImage(NULL, "bg.bmp", IMAGE_BITMAP, 640, 480, LR_LOADFROMFILE);
28  sheep = (HBITMAP)LoadImage(NULL, "sheep.bmp", IMAGE_BITMAP, 133, 220,
29      LR_LOADFROMFILE);
29  girl = (HBITMAP)LoadImage(NULL, "girl.bmp", IMAGE_BITMAP, 480, 148,
30      LR_LOADFROMFILE);
30  skill = (HBITMAP)LoadImage(NULL, "skill.bmp", IMAGE_BITMAP, 74, 60,
31      LR_LOADFROMFILE);
31  slash = (HBITMAP)LoadImage(NULL, "slash.bmp", IMAGE_BITMAP, 196, 162,
32      LR_LOADFROMFILE);
32  magic = (HBITMAP)LoadImage(NULL, "magic.bmp", IMAGE_BITMAP, 200, 100,
33      LR_LOADFROMFILE);
33  recover = (HBITMAP)LoadImage(NULL, "recover.bmp", IMAGE_BITMAP, 300, 150,
34      LR_LOADFROMFILE);
34  game = (HBITMAP)LoadImage(NULL, "over.bmp", IMAGE_BITMAP, 289, 74,
35      LR_LOADFROMFILE);
35
36  player.nHp = player.fHp = 50; //设定玩家角色生命值及上限
37  player.lv = 2; //设定玩家角色等级
38  player.w = 4; //设定攻击伤害加权值
39
40  monster.nHp = monster.fHp = 100; //设定怪物角色生命值及上限
41  monster.lv = 1; //设定怪物角色等级
42  monster.w = 1; //设定攻击伤害加权值
43
44  txtNum = 0; //显示消息数目
45
46  MyPaint(hdc);

```

```

47
48 return TRUE;
49 }

```

#### 程序说明

(1) 第 27~34 行: 加载图案到各个位图对象中。

(2) 第 36~38 行: 设定玩家角色生命值 (player.nHp)、生命值上限 (player.fHP)、等级和攻击伤害加权值。

(3) 第 40~42 行: 设定怪物角色生命值 (monster.nHp)、生命值上限 (monster.fHP)、等级和攻击伤害加权值。

#### 程序代码: WndProc()

```

1  //****消息处理函数*****
2  // 判断玩家是否下达攻击命令
3  LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
    lParam)
4  {
5      int x,y;
6
7      switch (message)
8      {
9          case WM_KEYDOWN:                //按键按下消息
10             if(wParam==VK_ESCAPE)        //按下【Esc】键
11                 PostQuitMessage(0);
12             break;
13          case WM_LBUTTONDOWN:            //单击鼠标左键消息
14             if(!attack)
15             {
16                 x = LOWORD(lParam);      //取得鼠标 X 坐标
17                 y = HIWORD(lParam);      //取得鼠标 Y 坐标
18
19                 if(x >= 500 && x <= 574 && y >= 350 && y <= 380)
20                     attack = true;
21             }
22             break;
23          case WM_DESTROY:                //窗口结束消息
24             DeleteDC(mdc);
25             DeleteDC(bufdc);
26             DeleteObject(bg);
27             DeleteObject(sheep);
28             DeleteObject(girl);
29             DeleteObject(skill);
30             DeleteObject(slash);
31             DeleteObject(magic);
32             DeleteObject(recover);
33             DeleteObject(game);
34
35             ReleaseDC(hWnd, hdc);

```



```

36
37         PostQuitMessage(0);
38         break;
39         default:                                //其他消息
40             return DefWindowProc(hWnd, message, wParam, lParam);
41     }
42     return 0;
43 }

```

## 程序说明

- (1) 第 13 行: 在消息处理函数中加入处理 “WM\_LBUTTONDOWN” (单击鼠标左键) 消息。
- (2) 第 16~17 行: 当玩家按下鼠标键时, 取得按下位置的 X、Y 轴坐标值并存储于 x、y 变量中。
- (3) 第 19~20 行: 判断 x 与 y 的值是否介于攻击命令图标的范围中, 若是则表示玩家按下了攻击命令, 之后将布尔变量 “attack” 的值设为 “true”, 开始一回合玩家与怪物的对战。

## 程序代码: MyPaint()

```

1  //****自定义绘图函数*****
2  // 1.画面贴图与对战信息显示
3  // 2.怪物行为判断及各项数值处理与计算
4  void MyPaint(HDC hdc)
5  {
6      char str[100];
7      int i,damage;
8
9      //贴上背景图
10     SelectObject(bufdc,bg);
11     BitBlt(mdc,0,0,640,480,bufdc,0,0,SRCCOPY);
12
13     //显示对战消息
14     for(i=0;i<txtNum;i++)
15         TextOut(mdc,0,360+i*18,text[i],strlen(text[i]));
16
17     //贴上怪物图
18     if(monster.nHp>0)
19     {
20         SelectObject(bufdc,sheep);
21         BitBlt(mdc,70,180,133,110,bufdc,0,110,SRCAHD);
22         BitBlt(mdc,70,180,133,110,bufdc,0,0,SRCPAINT);
23         sprintf(str,"%d / %d",monster.nHp,monster.fHp);
24         TextOut(mdc,100,320,str,strlen(str));
25     }
26
27     //贴上玩家图
28     if(player.nHp>0)
29     {
30         SelectObject(bufdc,girl);
31         BitBlt(mdc,500,200,60,74,bufdc,pNum*60,74,SRCAHD);

```

```

32     BitBlt(mdc,500,200,60,74,bufdc,pNum*60,0,SRCPAINT);
33     sprintf(str,"%d / %d",player.nHp,player.fHp);
34     TextOut(mdc,510,320,str,strlen(str));
35 }
36
37 if(over)                //贴上游戏结束图标
38 {
39     SelectObject(bufdc,game);
40     BitBlt(mdc,200,200,289,37,bufdc,0,37,SRCAHD);
41     BitBlt(mdc,200,200,289,37,bufdc,0,0,SRCPAINT);
42 }
43 else if(!attack)        //贴上攻击命令图标
44 {
45     SelectObject(bufdc,skill);
46     BitBlt(mdc,500,350,74,30,bufdc,0,30,SRCAHD);
47     BitBlt(mdc,500,350,74,30,bufdc,0,0,SRCPAINT);
48 }
49 else
50 {
51     f++;
52
53     //第5~10个画面时显示玩家攻击图标
54     if(f>=5 && f<=10)
55     {
56         SelectObject(bufdc,slash);
57         BitBlt(mdc,100,160,98,162,bufdc,98,0,SRCAHD);
58         BitBlt(mdc,100,160,98,162,bufdc,0,0,SRCPAINT);
59
60         //第10个画面时计算怪物受伤害程度并加入显示消息
61         if(f == 10)
62         {
63             damage = rand()%10 + player.lv*player.w;
64             monster.nHp -= (int)damage;
65
66             sprintf(str,"玩家攻击...玩家对怪物造成了 %d 的伤害。",damage);
67             MsgInsert(str);
68
69             CheckDie(monster.nHp,false);
70         }
71     }
72
73     srand(tPre);
74
75     //第15个画面时判断怪物进行哪项动作
76     if(f == 15)
77     {
78         if(monster.nHp > 20)                //生命值大于20
79         {
80             if(rand()%10 != 1)

```

```

81         monster.kind = 0;
82     else
83         monster.kind = 1;
84     }
85     else //生命值小于20
86     {
87         switch(rand()%5)
88         {
89             case 0: //普通攻击
90                 monster.kind = 0;
91                 break;
92             case 1: //施放攻击魔法
93                 monster.kind = 1;
94                 break;
95             case 2: //使尽全力攻击
96                 monster.kind = 2;
97                 break;
98             case 3: //补血
99                 monster.kind = 3;
100                break;
101             case 4: //逃跑
102                 monster.kind = 4;
103                 break;
104         }
105     }
106 }
107
108 //第26~30个画面时显示玩家攻击图标
109 if(f>=26 && f<=30)
110 {
111     switch(monster.kind)
112     {
113         case 0: //普通攻击
114             SelectObject(bufdc,slash);
115             BitBlt(mdc,480,150,98,162,bufdc,98,0,SRCAUD);
116             BitBlt(mdc,480,150,98,162,bufdc,0,0,SRCPAINT);
117
118             //第30个画面时计算玩家受伤程度并加入显示消息
119             if(f == 30)
120             {
121                 damage = rand()%10 + monster.lv*monster.w;
122                 player.nHp -= (int)damage;
123
124                 sprintf(str,"怪物攻击...怪物对玩家造成了 %d 的伤害。",damage);
125                 MsgInsert(str);
126
127                 CheckDie(player.nHp,true);
128             }
129             break;

```

```
130         case 1: //施放攻击魔法
131             SelectObject(bufdc,magic);
132             BitBlt(mdc,480,190,100,100,bufdc,100,0,SRCAHD);
133             BitBlt(mdc,480,190,100,100,bufdc,0,0,SRCPAINT);
134
135             //第30个画面时计算玩家受伤害程度并加入显示消息
136             if(f == 30)
137             {
138                 damage = rand()%10 + 3*monster.w;
139                 player.nHp -= (int)damage;
140
141                 sprintf(str,"怪物魔法攻击...怪物对玩家造成了 %d 的伤害。",
142                     damage);
143                 MsgInsert(str);
144
145                 CheckDie(player.nHp,true);
146             }
147             break;
148         case 2: //使尽全力攻击
149             SelectObject(bufdc,slash);
150             BitBlt(mdc,480,150,98,162,bufdc,98,0,SRCAHD);
151             BitBlt(mdc,480,150,98,162,bufdc,0,0,SRCPAINT);
152
153             //第30个画面时计算玩家受伤害程度并加入显示消息
154             if(f == 30)
155             {
156                 damage = rand()%10 + monster.lv*monster.w*5;
157                 player.nHp -= (int)damage;
158
159                 sprintf(str,"怪物全力攻击...怪物对玩家造成了 %d 的伤害。",
160                     damage);
161                 MsgInsert(str);
162
163                 CheckDie(player.nHp,true);
164             }
165             break;
166         case 3: //补血
167             SelectObject(bufdc,recover);
168             BitBlt(mdc,60,160,150,150,bufdc,150,0,SRCAHD);
169             BitBlt(mdc,60,160,150,150,bufdc,0,0,SRCPAINT);
170
171             //第30个画面时怪物恢复生命值并加入显示消息
172             if(f == 30)
173             {
174                 monster.nHp += 30;
175
176                 sprintf(str,"怪物魔法补血...恢复了 30 生命值。",damage);
177                 MsgInsert(str);
178             }
```

```

177             break;
178         case 4:
179             //第30个画面时判断怪物是否逃跑成功
180             if(f == 30)
181             {
182                 if(rand()%3 == 1)    //逃跑成功几率为1/3
183                 {
184                     over = true;
185                     monster.nHp = 0;
186
187                     sprintf(str, "怪物逃跑...逃跑成功。");
188                     MsgInsert(str);
189                 }
190                 else
191                 {
192                     sprintf(str, "怪物逃跑...逃跑失败。");
193                     MsgInsert(str);
194                 }
195             }
196             break;
197         }
198     }
199
200     if(f == 30)    //回合结束
201     {
202         attack = false;
203         f = 0;
204     }
205 }
206
207 BitBlt(hdc, 0, 0, 640, 480, mdc, 0, 0, SRCCOPY);
208
209 tPre = GetTickCount();
210
211 pNum++;
212 if(pNum == 8)
213     pNum = 0;
214 }

```

## 程序说明

MyPaint()函数主要用来进行所有图案及对战消息的贴图操作，其中利用变量“f”计算每一回合中程序更新窗口画面的次数，并在固定画面更新的区间中重复贴上攻击效果的图案，产生画面延迟的效果。这样才不会让程序因为画面更新速率过快而使得攻击效果显示的时间过于短暂，而无法达到一般视觉上的要求。

(1) 第14~15行：以目前变量“txtNum”中记录的显示消息数目为临界值，使用循环在画面左下方输出对战消息。

(2) 第18~25行：当怪物目前的生命值(monster.nHp)大于“0”时，表示怪物还存在，则进行怪物在背景画面上的透明贴图。第23、24行将怪物目前的生命值及最大生命值转换成字符串，

显示在图标下方。

(3) 第 28~35 行: 当玩家目前的生命值 (`player.nHp`) 大于“0”时, 进行玩家人物在背景画面上的透明贴图, 并在图标下方显示玩家目前的生命值及最大生命值。

(4) 第 37~42 行: 当程序中玩家或者怪物某一方的生命值降至“0”以下, “over”变量设为“true”, 表示对战结束。第 39~41 行程序代码在画面上进行对战结束图案的透明贴图。

(5) 第 43~48 行: 当 over 不为 true 时, 表示对战尚在进行当中, 此时如果“attack”变量不为“true”, 表示玩家尚未按下攻击命令。第 45~48 行程序代码进行攻击命令的透明贴图。

(6) 第 49~205 行: 当“attack”为“true”时, 表示玩家已按下了攻击命令进行回合的对战, 接下来的程序代码会进行玩家攻击贴图、怪物反击时的决策判断及怪物的行为贴图。

(7) 第 51 行: 递增“f”变量的值, “f”变量的值在每一回合结束时会重设为“0”, 这样每回合开始时其值为“0”。

(8) 第 54~58 行: 在玩家按下攻击命令后, 在第 5~10 个画面时进行玩家攻击怪物的透明贴图, 因此玩家攻击时可看到 6 个画面的攻击暂留效果。

(9) 第 61~70 行: 在回合开始后的第 10 个画面时 (玩家攻击怪物画面效果结束时), 计算怪物受到的伤害并重设怪物现有的生命值。调用 `MsgInsert()` 函数加入这一攻击消息到“text”中。最后调用 `CheckDie()` 函数检查怪物的生命值是否降至“0”以下。

(10) 第 63~64 行: 第 63 行按照玩家的等级与加权值计算玩家对怪物所造成的伤害并存储在“damage”变量中。第 64 行则是按照玩家所造成的伤害, 减少怪物的生命值 (`monster.nHP`)。这里使用的伤害计算公式如下:

$$\text{rand}()\%10 + \text{player.lv} * \text{player.w}$$

公式中, “`rand()%10`”会产生一个 0~9 间的数值, 以这个数值与“`player.lv*player.w`”的计算结果相加, 可以产生一个“`player.lv*player.w~player.lv*player.w+9`”之间的攻击伤害数值。在这个公式中, 当玩家的等级“`player.lv`”或者玩家攻击伤害的加权值“`player.w`”越高时, 造成的攻击伤害程度也越大。这样的计算方式也适用于一般 RPG 游戏中随着玩家等级或者能力的提高, 攻击伤害程度越大的模式。

(11) 第 69 行: 调用 `CheckDie()` 函数检查怪物受到攻击后生命值是否降至“0”以下。输入的第 1 个参数为怪物的生命值“`monster.nHP`”。第 2 个输入参数为“false”, 表示要判断的对象为怪物 (非玩家)。

(12) 第 73 行: 因为接下来的程序会多次调用 `rand()` 函数来产生随机数, 所以这行程序代码用来调用 `srand()` 函数设定的随机数种子。其中输入的参数为“`tPre`”, “`tPre`”记录上次画面更新的时间, 因此会随着程序运行时间的改变而改变, 所以设定“`tPre`”为随机数种子便可在后面产生顺序不固定的随机数 (如果未设定变动的随机数种子, `rand()` 函数将会产生一组顺序固定的随机数)。

(13) 第 76~106 行: 在回合开始后的第 15 个画面时, 判断怪物要进行何种行为。其中第 78~105 行程序代码实际上是前面所看过的怪物对战行为算法, 这里以一个“kind”结构成员记录怪物会进行何种行为, “kind”的值有“0”、“1”、“2”、“3”和“4”, 分别表示怪物进行普通攻击、魔法攻击、使尽全力攻击、补血和逃跑。

(14) 第 109~198 行: 判断出怪物在回合中要进行的行后, 接下来在回合中的第 26~30 个画面间进行与怪物行为相关的贴图与计算。

(15) 第 113~129 行: 当“`monster.kind`”的值为“0”时表示怪物进行普通攻击, 第 114~117 行程序代码会贴上普通攻击的图标。而运行到第 30 个画面时, 第 119~128 行程序代码以怪物的等级与攻击伤害加权值来计算玩家所受到的伤害, 接着减少玩家的生命值, 调用 `MsgInsert()` 函数加入

攻击消息并调用 CheckDie()函数检查玩家的生命值是否降至“0”以下。

(16) 第 130~146 行: 当“monsert.kind”的值为“1”时表示怪物进行魔法攻击, 第 136~145 行程序代码会贴上魔法攻击的图标。同样地当运行到第 30 个画面时, 进行玩家受到伤害的相关计算、加入对战消息与检查玩家生命值的动作。

(17) 第 147~163 行: 当“monsert.kind”的值为“2”时表示怪物进行全力攻击, 在这里运行的程序代码与前面普通攻击相同, 而在计算玩家受到怪物的攻击伤害时, 则以怪物等级与加值权相乘, 再乘上“5”来增加攻击的伤害力。

(18) 第 164~177 行: 当“monster.kind”的值为“3”时表示怪物进行补血, 第 165~168 行程序代码会贴上补血的图标。运行到第 30 个画面时, 将怪物的生命值恢复 30 个单位并调用 MsgInsert()函数加入动作消息。

(19) 第 178~196 行: 当“monster.kind”的值为“4”时表示怪物进行逃跑, 则不需进行贴图的操作。运行到第 30 个画面时, 以随机数决定怪物是否逃跑成功, 若逃跑成功则将“over”变量设为“true”, 怪物的生命值“monster.nHP”设为“0”, 并加入怪物逃跑成功的消息。这样在下次画面更新时, 怪物将不显示, 同时按照第 37 行程序代码的判断会显示游戏结束图案。

(20) 第 200~204 行: 由于在这个范例中设定一回合的对战以 30 个画面来完成。当 f 变量的值为“30”时, 便已经完成了一回合的所有动作。接下来将“f”变量的值设为“0”, “attack”变量的值设为“false”, 准备由玩家下达攻击命令进行下一回合的对战。

## 程序代码: MsgInsert()

```
1  //****对战消息新增函数*****
2  void MsgInsert(char* str)
3  {
4      if(txtNum < 5)
5      {
6          sprintf(text[txtNum],str);
7          txtNum++;
8      }
9      else
10     {
11         for(int i=0;i<txtNum;i++)
12             sprintf(text[i],text[i+1]);
13
14         sprintf(text[4],str);
15     }
16 }
```

## 程序说明

MsgInsert()函数是用来将对战消息加入到“text”字符串数组中, 函数的输入参数“str”是一个字符串数组的指针 (即某一对战消息字符串)。

(1) 第 4~8 行: 当目前对战消息总数“txtNum”小于“5”时, 将输入的消息字符串加以转换, 并直接以“txtNum”为索引值把消息字符串加入到“text”中。

(2) 第 9~15 行: 当目前对战消息总数“txtNum”不小于“5”时, 表示此时消息数目已经超过上限。因此第 11~12 行程序代码以循环将字符串数组中的每一字符串消息向前移动, 从而删除在最前面的字符串。接着第 14 行程序代码将要新增的字符串加入到字符串数组的最末端。

## 程序代码: CheckDie()

```

1  /***玩家怪物生命值判断函数***/
2  void CheckDie(int hp,bool player)
3  {
4      char str[100];
5
6      if(hp <= 0)
7      {
8          over = true;
9          if(player)
10         {
11             sprintf(str,"你被怪物打败了...");
12             MsgInsert(str);
13         }
14         else
15         {
16             sprintf(str,"怪物歼灭...");
17             MsgInsert(str);
18         }
19     }
20 }

```

## 程序说明

在这个范例中,当怪物或者玩家进行攻击后,会调用 CheckDie()函数来检查对方的生命值是否降到0以下,一旦有任何一方的生命值降至0以下便代表游戏结束。其中第6行程序代码判断输入的生命值是否小于“0”,若小于“0”则先将“over”变量设为“true”,接着再判断是玩家或是怪物死亡,之后调用 MsgInsert()函数加入相关的显示消息。

## 运行结果

程序运行结果如图 5-5~图 5-8 所示。



玩家进行普通攻击

图 5-5



怪物进行魔法攻击

图 5-6





怪物进行补血

图 5-7



任一方生命值降至0以下，则游戏结束

图 5-8

以上介绍了行为型游戏 AI 的基本设计原理，并介绍了一个怪物和玩家对战的范例程序。接下来的小节将继续探讨同样以行为型 AI 原理为基础的计算机搜寻迷宫出口的游戏。

## 5.2.2 搜寻迷宫出口

计算机搜寻迷宫出口的游戏与行为型 AI 有直接的关系，在搜寻迷宫出口的过程中，计算机必须对于接下来该往哪一个方向移动做思考判断，这与前一小节中怪物接下来该进行哪一种行为的意义是一样的。

不过在搜寻迷宫出口游戏中，除判断下一时刻移动的目的外，计算机角色还必须对走过的迷宫路径做记录，这样才可以让计算机角色走到迷宫中的死路时可以回头搜寻其他路径。

这一小节中会设计一个小球在迷宫中移动搜寻出口的范例程序，其中用到了行为型 AI 及地图拼接的概念，并利用堆栈 (stack) 数据结构来存储搜寻迷宫时所经过的每个迷宫的方格编号。下面一来讨论这些技巧在搜寻迷宫出口游戏中的应用。

### 1. 迷宫拼接贴图

搜寻迷宫出口的第一件工作就是建立一个迷宫并显示在窗口中，在这里以第 2 章中所讨论过的平面地图拼接法来产生程序中的迷宫。范例中是以一个一维数组来定义迷宫的内容。

```
const int rows = 8, cols = 8; //定义迷宫的列数与行数
int mapIndex[rows*cols]={0,2,0,0,0,0,0,0, //第1列
                        0,1,0,1,1,1,1,0, //第2列
                        0,1,0,1,0,1,1,0, //第3列
                        0,1,0,0,0,1,1,0, //第4列
                        0,1,1,1,1,1,1,0, //第5列
                        0,1,0,0,0,0,1,0, //第6列
                        0,0,1,1,1,1,1,0, //第7列
                        0,0,0,0,0,0,3,0 }; //第8列
```

将“mapIndex[]”一维数组编排成行列的格式，其中每个数组元素事实上就代表着平面迷宫中的每一个迷宫方格，元素值“0”、“1”、“2”和“3”分别代表迷宫的墙、通道、入口和出口。图 5-9

是以“mapIndex[]”这个数组进行拼接贴图后所产生的迷宫外观，其中每个迷宫方格上标出了该方格对应的数组元素值。

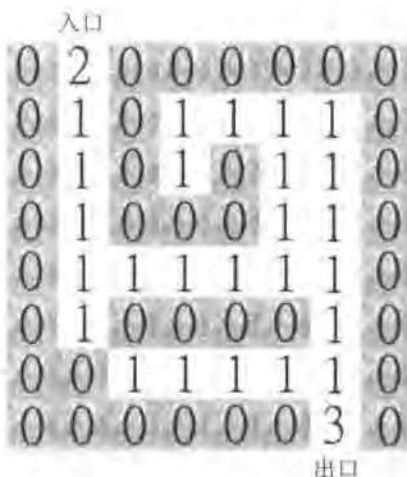


图 5-9

按照“mapIndex[]”数组内容拼接完成后的迷宫，方格排列会与数组定义时的行列排列顺序相同，而迷宫方格的编号就是其对应数组元素的索引值。例如图 5-9 中最上面的一列，其方格编号由左到右分别是“0”、“1”、“2”、……、“7”。

在定义迷宫数组时，将代表墙的数组元素值设为“0”，其原因是这样可以很容易地利用下面的判断式来判断某一方格是否可通过。

```
if(mapIndex[编号])
    //可通过，元素值为“1”、“2”或“3”
else
    //不可通过，元素值为“0”
```

## 2. 迷宫搜寻规则

了解了迷宫数组的定义与拼接贴图的方法之后，接着要来讨论搜寻迷宫出口的规则。这里的规则，指的就是计算机的行为型 AI，即下一步该往哪里走？没路了怎么办？

对于迷宫出口的搜寻问题，并没有什么特别的方式可以预测出口在哪里，因此搜寻的算法必须让计算机探索迷宫中的每一块区域来找寻出口。范例中所使用的搜寻规则，是当每次计算机控制的小球要进行移动时，判断目前所在位置上、下、左、右方的通道方格是否还没走过，如果没有走过就往该方格移动。当小球走到无路可走时（四周的方格是墙或者已走过），就往回走一格，接着再按照相同的方式判断四周是否有没走过的通道方格。按照这样的搜寻规则不断进行，直到找到出口，或者退回到原点表示迷宫没有出口为止。

上面这样一个迷宫搜寻的概念，可以利用下面的算法来描述。

```
1  if(上一格可走)
2  {
3      加入方格编号到堆栈;
4      往上走;
5      判断是否为出口;
6  }
```

```

7  else if(下一格可走)
8  {
9      加入方格编号到堆栈;
10     往下走;
11     判断是否为出口;
12 }
13 else if(左一格可走)
14 {
15     加入方格编号到堆栈;
16     往左走;
17     判断是否为出口;
18 }
19 else if(右一格可走)
20 {
21     加入方格编号到堆栈;
22     往右走;
23     判断是否为出口;
24 }
25 else
26 {
27     从堆栈删除一方格编号;
28     从堆栈中取出一方格编号;
29     往回走;
30 }

```

上面的算法是每次进行移动时所运行的内容，主要用来判断目前所在位置的上、下、左、右是否有可以前进的方格，若找到可移动的方格，便将该方格的编号加入到记录移动路径的堆栈中，并往该方格移动。当四周没有可走的方格时（第 25 行），也就是目前所在的方格无法走出迷宫时，必须退回到前一格重新检查是否有其他可走的路径，所以在上面算法中的第 27 行会将目前所在位置的方格编号从堆栈中删除，之后第 28 行再取出的就是前一次所走过的方格编号。

### 3. 堆栈结构的使用

在搜寻迷宫出口的算法中使用了一个堆栈数据结构，用来记录走过的方格编号。下面介绍有关堆栈在搜寻迷宫出口游戏中的运用方式。

堆栈是一种只能在最末端进行数据存取的数据存储结构。在搜寻迷宫出口的范例程序中，当走到某一未走过的方格时，会将方格编号加入到堆栈当中，下面利用图 5-10 来做说明。在这里假设原先的堆栈中已经加入了几个方格编号，而此时要加入新的方格编号为 11。



图 5-10

图中显示了堆栈结构及加入方格编号到堆栈后堆栈内容的变化。堆栈有如图中所示的“末端指针”，这个指针永远都是指向堆栈中最后一个元素，上图中的堆栈在加入了一个新的编号之后，末端指针便往后移动一格，指向最后一个元素。由这样的一个特性来看，将堆栈应用在搜寻迷宫出口的堆栈时，由于每次新加入的方格编号必定会在堆栈的最末端，因此堆栈末端指针所指的方格编号便是目前搜寻迷宫出口的小球所在的位置。

除了末端指针指向目前位置所在方格编号的这项特性外，利用堆栈还可以在搜寻迷宫发现无路可走时，退回到之前走过的方格重新搜寻其他可行的路径。延续上一个例子，现在假设前面走到编号为 11 的方格时发现已无路可走，此时可直接由堆栈末端删除最后这个走不出迷宫的错误方格编号，这样堆栈的末端指针将会指向前一次走过的方格，这样便可以退回到上一方格再来重新检查是否有其他未走过的路径。如图 5-11 所示说明了从记录搜寻路径堆栈中删除方格编号的操作及其内容的变化。



图 5-11

堆栈的删除操作是直接删除末端指针所指的元素，再将末端指针指向堆栈的最后一个元素。图 5-11 中，当从堆栈中删除了编号 11 的方格之后，末端指标则指向了编号 14 的方格，这样迷宫的搜寻路径便可退回到前一个编号 14 的方格上。

以上就是有关堆栈结构在搜寻迷宫出口游戏中的应用。范例中直接利用了 C++ 标准函数库的定义来建立程序所需要的堆栈并进行了各项相关操作，下面来看看这个搜寻迷宫出口 AI 的范例程序内容。

## 《范例》 ch5\_3

利用拼接贴图的方法产生迷宫，并利用堆栈记录搜寻路径，设计搜寻迷宫出口行为型 AI。

### 程序代码：引用头文件

```
1 //引用头文件
2 #include "stdafx.h"
3 #include <stack>
4 using namespace std;
```

### 程序说明

第 4~5 行：引用 C++ 标准函数库中的“stack”头文件。此头文件中定义了“stack”类型说明，可用来建立存储各种不同数据类型的堆栈结构。

### 程序代码：常数定义与全局变量声明

```
1 //常数定义
2 const int rows = 8, cols = 8;
3
```

```

4 //全局变量声明
5 HINSTANCE hInst;
6 HBITMAP ball;
7 HDC hdc,mdc,bufdc;
8 HWND hWnd;
9 DWORD tPre,tNow;
10 char *str;
11 int nowPos,prePos;
12 bool find;
13 stack<int> path;
14
15 int mapIndex[rows*cols] = { 0,2,0,0,0,0,0,0, //第1列
16                             0,1,0,1,1,1,1,0, //第2列
17                             0,1,0,1,0,1,1,0, //第3列
18                             0,1,0,0,0,1,1,0, //第4列
19                             0,1,1,1,1,1,1,0, //第5列
20                             0,1,0,0,0,0,1,0, //第6列
21                             0,0,1,1,1,1,1,0, //第7列
22                             0,0,0,0,0,0,3,0 }; //第8列
23 int record[rows*cols];

```

## 程序说明

(1) 第 2 行：定义常数“rows”与“cols”，分别代表迷宫方格的行数与列数，其默认值都是“8”。可通过改变这两个常数的值来设定迷宫的大小。

(2) 第 10 行：记录目前搜寻状态的字符串指针。

(3) 第 11 行：声明变量“nowPos”用来记录小球目前位置的方格编号，“prePos”变量则是上一次位置的方格编号。程序中利用“nowPos”的值计算出此次小球的贴图位置并进行贴图，而利用“prePos”计算出上次小球的贴图位置，并以白色覆盖上次的小球贴图，以去除残留影像。

(4) 第 12 行：声明一布尔变量“find”，以记录是否找到迷宫出口。

(5) 第 13 行：由“stack”类型建立一整数型(int)堆栈“path”，用以记录移动路径。

(6) 第 15~22 行：建立“mapIndex[]”数组并定义迷宫的内容，元素值“0”、“1”、“2”和“3”分别代表迷宫的墙、通道、入口和出口。

(7) 第 23 行：声明一数组“record[]”，用来标记迷宫中不可通过的方格(墙)及已走过的方格。

## 程序代码：InitInstance()

```

1 //****初始化函数*****
2 // 1.迷宫拼接贴图
3 // 2.设定小球在迷宫入口处
4 // 3.按照“mapIndex”设定“record”数组的初始内容
5 BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
6 {
7     HBITMAP bmp;
8     hInst = hInstance;
9
10    hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
11        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

```

```
12
13 if (!hWnd)
14 {
15     return FALSE;
16 }
17
18 MoveWindow(hWnd,10,10,430,450,true);
19 ShowWindow(hWnd, nCmdShow);
20 UpdateWindow(hWnd);
21
22 hdc = GetDC(hWnd);
23 mdc = CreateCompatibleDC(hdc);
24 bufdc = CreateCompatibleDC(hdc);
25
26 bmp = CreateCompatibleBitmap(hdc,cols*50,rows*50);
27 SelectObject(mdc,bmp);
28
29 HBITMAP tile;
30 int rowNum,colNum;
31 int i,x,y;
32
33 tile = (HBITMAP)LoadImage(NULL,"tile.bmp",IMAGE_BITMAP,50,50,LR_LOADFROMFILE);
34 ball = (HBITMAP)LoadImage(NULL,"ball.bmp",IMAGE_BITMAP,50,50,LR_LOADFROMFILE);
35
36 //按照“mapIndex”数组中的定义进行迷宫拼接
37 for (i=0;i<rows*cols;i++)
38 {
39     record[i] = mapIndex[i];
40
41     rowNum = i / cols;           //求列编号
42     colNum = i % cols;           //求行编号
43     x = colNum * 50;             //求贴图 X 坐标
44     y = rowNum * 50;             //求贴图 Y 坐标
45
46     SelectObject(bufdc,tile);
47
48     if(!mapIndex[i])             //墙
49         BitBlt(mdc,x,y,50,50,bufdc,0,0,SRCCOPY);
50     else                          //通道
51     {
52         if(mapIndex[i] == 2)     //找到迷宫出口
53         {
54             nowPos = i;
55             path.push(i);
56             record[i] = 0;
57         }
58         BitBlt(mdc,x,y,50,50,bufdc,0,0,WHITENESS);
59     }
60 }
```

```

61  prePos = cols * rows + 1;
62
63  MyPaint(hdc);
64
65  return TRUE;
66 }

```

## 程序说明

InitInstance()函数依 mapIndex[]数组的内容进行迷宫的拼接,并设定 record[]数组中的内容,同时将小球的位置设定在迷宫的入口处。

(1) 第 33~34 行:加载迷宫图块及小球图案。

(2) 第 37~60 行:以循环取出“mapIndex[]”数组中的各元素,进行图块的拼接、迷宫入口的判断,以及“record[]”数组中各元素初值的设定。

(3) 第 39 行:设定“record[]”数组的元素值。“record[]”数组的内容一开始与“mapIndex[]”相同,各元素的值分别记录着迷宫中的各个方格是否可通过。进行迷宫搜寻时,小球每走过一个方格,该方格在“record[]”中对应的元素值被设定为“0”,表示该方格已不可通过。这样才能在搜寻未走过的方格时排除这些已走过的区域,确保完整地搜寻整个迷宫。

(4) 第 41~59 行:按照第 2 章中所讨论过的地图拼接方法,先计算各个迷宫图块的行编号与列编号;然后再计算出实际的贴图坐标;接着再判断图块是迷宫的墙还是通道,判断为墙时,便贴上大小为 50×50 的迷宫墙图案(第 49 行),否则即表示为通道、迷宫入口或出口。第 58 行程序代码会进行相同的贴图操作,但要 will BitBlt()贴图函数中最后一个参数设为“WHITENESS”,这样所显示的才会是一个 50×50 的空白方格。

(5) 第 52~57 行:当判断“mapIndex[]”数组元素值为“2”时,表示该元素为迷宫的入口。接下来第 54 行程序代码将小球的所在位置“nowPos”设定在此元素值所代表的迷宫方格上。第 55 行程序代码将方格编号加入到堆栈“path”中。第 56 行程序代码则设定“record[]”数组中对应的元素值为“0”,表示这一方格已走过。

(6) 第 55 行:由“stack”类型产生的堆栈“path”,可以直接调用类别中所定义的函数成员来进行各项堆栈相关的操作,下面列出了最常用且在范例中使用的函数成员。

```

void push(元素);      //加入元素到堆栈
void pop();           //从堆栈删除元素
int size();           //堆栈大小
数据类型 top();       //取得堆栈末端元素

```

(7) 第 61 行:设定“prePos”的初值,在此设定为“cols×rows+1”,使得程序第 1 次进行窗口绘图时,在迷宫外进行空白贴图不影响画面显示的内容。之后窗口画面更新时,“prePos”记录着小球前次移动所在的方格编号,可用来计算上次小球的贴图坐标,并以空白加以覆盖清除。

## 程序代码: MyPaint()

```

1  //****自定义绘图函数*****
2  // 搜寻可行路径及小球移动贴图
3  void MyPaint(HDC hdc)
4  {
5      int rowNum,colNum;
6      int x,y;
7      int up,down,left,right;

```

```
8
9 //清除上次贴图
10 rowNum = prePos / cols;
11 colNum = prePos % cols;
12 x = colNum * 50;
13 y = rowNum * 50;
14
15 SelectObject(bufdc,ball);
16 BitBlt(mdc,x,y,50,50,bufdc,0,0, WHITENESS);
17
18 //小球贴图
19 rowNum = nowPos / cols;
20 colNum = nowPos % cols;
21 x = colNum * 50;
22 y = rowNum * 50;
23
24 SelectObject(bufdc,ball);
25 BitBlt(mdc,x,y,50,50,bufdc,0,0, SRCCOPY);
26
27 if(!find)
28 {
29     str = "找寻出口中...";
30
31     up = nowPos - cols;
32     down = nowPos + cols;
33     left = nowPos - 1;
34     right = nowPos + 1;
35
36     if(up>=0 && record[up]) //往上走
37     {
38         path.push(up);
39         record[up] = 0;
40         prePos = nowPos;
41         nowPos = up;
42
43         if(mapIndex[nowPos] == 3) //找到出口
44             find = true;
45     }
46     else if(down<=cols*rows-1 && record[down]) //往下走
47     {
48         path.push(down);
49         record[down] = 0;
50         prePos = nowPos;
51         nowPos = down;
52
53         if(mapIndex[nowPos] == 3)
54             find = true;
55     }
56     else if(left>=rowNum*cols && record[left]) //往左走
```



```

57     {
58         path.push(left);
59         record[left] = 0;
60         prePos = nowPos;
61         nowPos = left;
62
63         if(mapIndex[nowPos] == 3)
64             find = true;
65     }
66     else if(right <= (rowNum+1)*cols-1 && record[right])    //往右走
67     {
68         path.push(right);
69         record[right] = 0;
70         prePos = nowPos;
71         nowPos = right;
72
73         if(mapIndex[nowPos] == 3)
74             find = true;
75     }
76     else    //无可移动方格
77     {
78         if(path.size() <= 1)    //回到入口
79             str = "迷宫无出口...";
80         else
81         {
82             path.pop();
83             prePos = nowPos;
84             nowPos = path.top();
85         }
86     }
87 }
88 else
89 {
90     str = "找到出口了!";
91 }
92
93 TextOut(mdc,0,0,str,strlen(str));
94 BitBlt(hdc,10,10,cols*50,rows*50,mdc,0,0,SRCCOPY);
95
96 tPre = GetTickCount();
97 }

```

## 程序说明

在 MyPaint() 函数中, 先进行小球移动的贴图操作, 接着再搜寻下次画面更新时小球要往哪个方格移动。

- (1) 第 10~16 行: 按照 “prePos” 的值进行前次小球贴图清除操作。
- (2) 第 19~25 行: 按照 “nowPos” 的值进行此次小球移动的贴图操作。
- (3) 第 27 行: 范例中利用布尔变量 “find” 来记录是否已找到迷宫出口, 当 “find” 的值不为

“true”时，则进行接下来搜寻迷宫中未走过方格的判断操作。

(4) 第 31~34 行：要判断目前所在方格的四周是否有未走过的方格前，必须先计算出四周的方格编号，即以目前小球所在的方格为中心，其上 (up)、下 (down)、左 (left)、右 (right) 方格编号的计算。下面以图 5-12 进行说明。

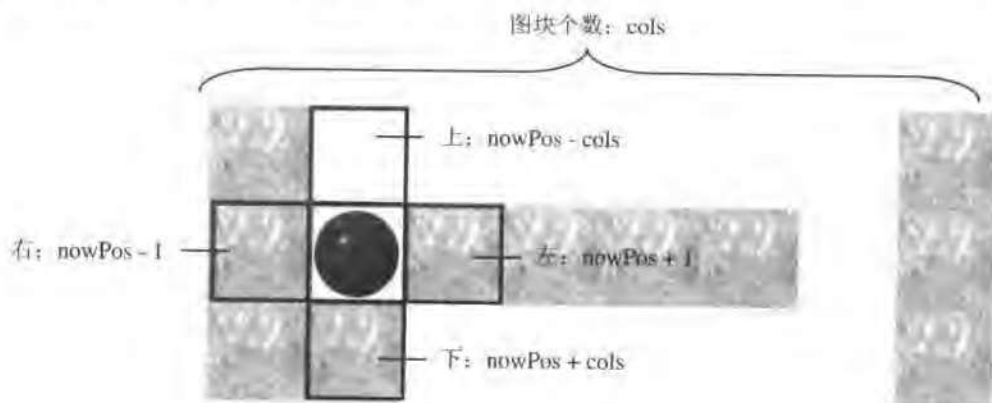


图 5-12

(5) 第 36、46、56、66 行：按顺序判断目前所在方格的上、下、左、右是否有未走过的通道，其中判断的条件式是按照“record[]”数组中的内容来判断方格可否通过以及前面是否已经走过，而且方格的编号必须落在该方向上的临界编号之内。临界编号是用来限定各个方向上的有效方格编号，当计算出目前所在位置四周方格的编号后，若某一方向上的方格编号超出了该方向上的临界编号，则表示不能往该方向移动。下面以图 5-13~图 5-16 进行说明。

(6) 第 36~45 行：当目前小球所在位置的上方方格编号“up”不超出临界编号，且为未走过的通道时 (record[up]==1)，小球便会向此方格移动，并检查这个方格是否是迷宫的出口。

(7) 第 38~41 行：在判断上方方格为未走过的通道后，第 38 行程序代码会先将方格编号加入到记录移动路径的堆栈“path”中。第 39 行程序代码设定“record[]”数组中方格对应的元素值为“0”，表示方格已经走过。第 40 行程序代码将小球目前所在的方格编号存储在“prePos”变量。最后第 41 行程序代码则重设小球位置“nowPos”为新的方格编号“up”。



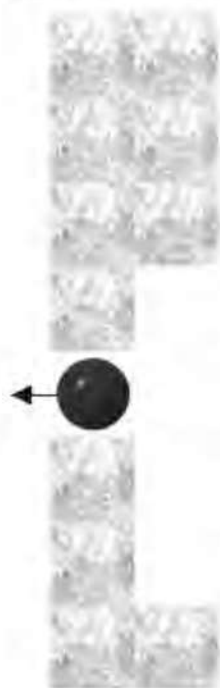
小球移动到最上面一列时，不能再往上移动，根据小球目前位置计算出的上方方格编号会超出临界编号“0” (数组最小索引值)。

图 5-13



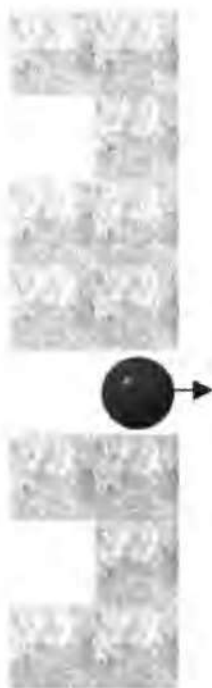
小球移动到最下面一列时，不能再往下移动，根据小球目前位置计算出的下方方格编号会超出临界编号“cols\*rows-1” (数组最大索引值)。

图 5-14



小球移动到最左边一行时，不能再往左移动，根据小球目前位置计算出的左方方格编号会超出临界编号“ $\text{rowNum} * \text{cols}$ ”（小球所在列的最小索引值）。

图 5-15



小球移动到最右边一行时，不能再往右移动，根据小球目前位置计算出的右方方格编号会超出临界编号“ $(\text{rowNum} + 1) * \text{cols} - 1$ ”（小球所在列的最大索引值）。

图 5-16

（8）第 43~44 行：在设定小球到达新的方格之后，判断方格在“`mapIndex[]`”中对应的元素值是否为“3”，若为“3”便表示此次移动已经找到了迷宫的出口。

（9）第 46~75 行：以相同的方法按照顺序判断其他方向上是否有未走过的通道。当发现新的可移动方格时，便进行如上面所讨论的小球位置更新与判断是否为出口的操作。

（10）第 76~86 行：在前面的判断中，如果发现四周无可移动的方格时，则小球必须往回走并搜寻其他可行路径。

（11）第 78~79 行：判断记录移动路径的堆栈中所存储的方格数“`path.size()`”是否小于等于“1”。如果其中记录走过的方格数小于等于“1”时，表示堆栈中仅包含最初迷宫入口处的方格，也就是说经过之前的搜寻小球已退回到了出口处，迷宫并无出口。

（12）第 82~84 行：当堆栈中记录的方格数大于“1”时，进行重设小球使其回到前一方格的操作。第 82 行程序代码调用 `pop()` 函数从“`path`”堆栈中删除目前所在位置的方格编号。第 83 行程序代码记录小球现在的所在位置到“`prePos`”变量中。第 84 行程序代码则在完成前面删除操作之后，调用 `top()` 函数取出堆栈中最后一个元素值，即小球上一次所走过的方格编号，并以此编号重设小球的所在位置。

（13）第 90 行：当“`find`”变量为“`true`”时，运行此行程序代码，此时由于小球已找到迷宫出口，便不再进行任何操作，仅设定已找到迷宫出口的字符串消息。

（14）第 93~94 行：输出字符串消息，以及在前面完成小球移动的贴图操作后，进行窗口画面的更新。

### 运行结果

程序运行结果如图 5-17 和图 5-18 所示。



图 5-17



图 5-18

这一节中介绍了行为型游戏 AI 的设计概念。行为型 AI 可以说是游戏中最常用的一种 AI 模式，而由这样一种 AI 模式所拓展出的游戏 AI 架构所涵盖的范围可说是相当广泛。下一节中将探讨以行为型 AI 的条件判断方式及数学计算来求出计算机最佳决策的策略型游戏 AI。

### 5.3 策略型游戏 AI

策略型游戏 AI 可以说是游戏 AI 中比较复杂的一种，最常见的运用策略型 AI 的游戏是棋盘式游戏。在这一类型的游戏中，计算机必须按照目前的状况来判断所有可走的棋步与可能的获胜状况，并在每一回合中计算盘面上计算机或者玩家的可能获胜几率，以决定最佳的走法。

这一节里将讨论策略型 AI 中计算机的决策方式，并以一个五子棋范例将策略型 AI 运用于游戏中。

“当局者迷，观局者清”，这句话用在由计算机所控制的玩家上是不成立的。相反，计算机必须在每回合下棋时都知道有哪些获胜的方式，并计算出每下一步棋到棋盘任一格子上的获胜几率。接下来，我们将以计算机与玩家进行五子棋对弈的一个游戏范例来说明策略型游戏 AI 的设计方式。

#### 1. 求得所有获胜组合

在一场五子棋的棋局中，计算机必须要知道有哪些获胜的组合。事实上这些组合就是用来判断计算机或者玩家双方是否有任一方已经有获胜的条件。稍后可看到程序是利用数组来存储这些获胜组合的，并在计算机或者玩家每下一步棋时，通过修改数组中的内容判断计算机或玩家是否已完成某一获胜的组合而赢得棋局。

范例使用了  $10 \times 10$  大小的五子棋盘，下面以图 5-19~图 5-22 来说明棋盘上可能获胜的组合并计算这些组合的总数。

(1) 水平方向上的获胜组合数如图 5-19 所示。

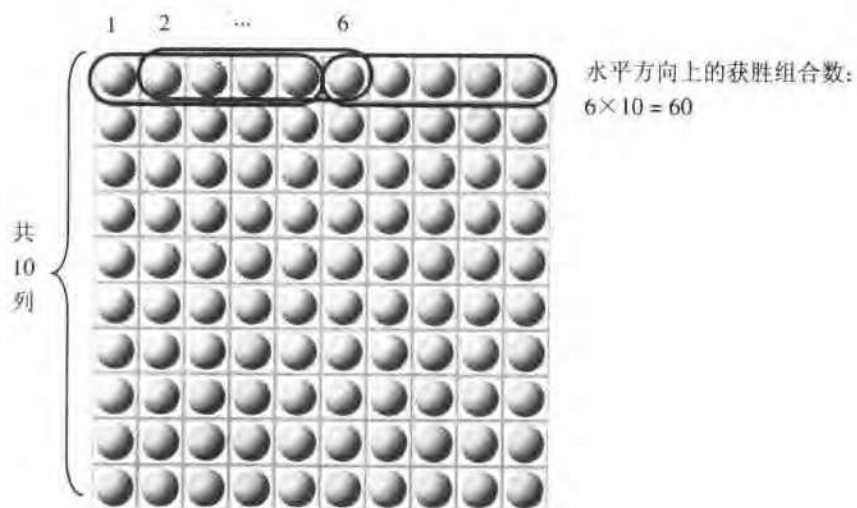


图 5-19

(2) 垂直方向上的获胜组合数如图 5-20 所示。

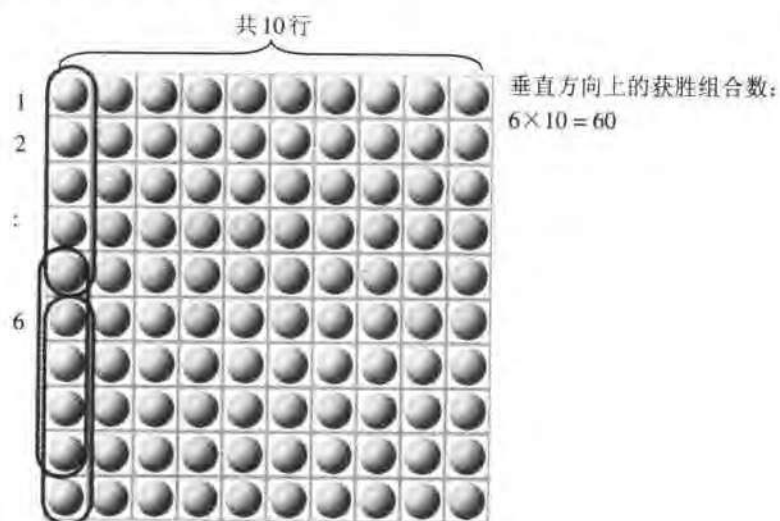
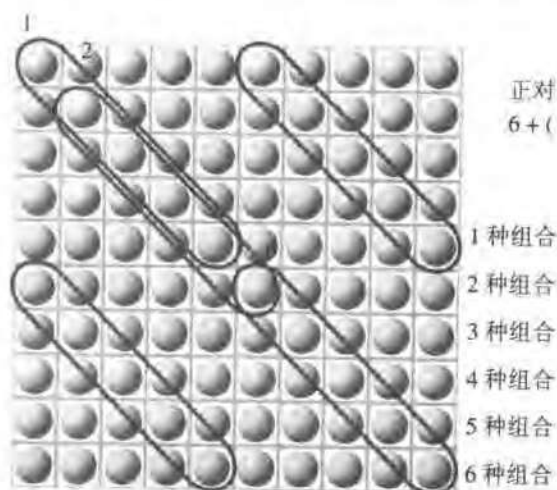


图 5-20

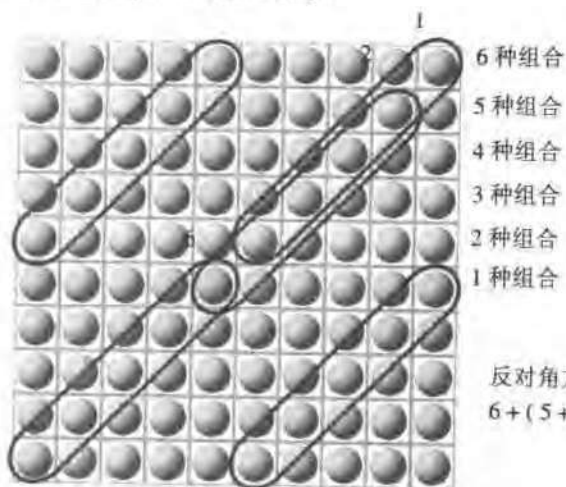
(3) 正对角方向上的获胜组合如图 5-21 所示。



正对角方向上的获胜组合数：  
 $6 + (5 + 4 + 3 + 2 + 1) \times 2 = 36$

图 5-21

(4) 反对角方向上的获胜组合数如图 5-22 所示。



反对角方向上的获胜组合数：  
 $6 + (5 + 4 + 3 + 2 + 1) \times 2 = 36$

图 5-22

通过几个方向上的计算，可以求出一个  $10 \times 10$  的五子棋盘共有 192 种获胜的组合。算出获胜组合的总数之后，下面来说明程序是如何以数组的形式来存储这些获胜组合的，并通过建立获胜表来设计计算机 AI 以及作为棋局胜负判定的参考。

## 2. 建立与使用获胜表

在说明程序所使用的获胜表之前，首先来看看程序中对于棋盘上棋格位置的表示方法。范例利用  $10 \times 10$  的二维数组来记录所有棋格的位置。每个棋格都是以数组的行列编号（元素索引值）来表示的。图 5-23 便是以这样的方式来标示位于棋格上棋子的位置。

	0	1	2	3	4	5	6	7	8	9	行编号
0	v115			[1][3]		[1][5]					
1				●		●					
2								[2][7]			
3											

图 5-23

了解棋盘上棋子位置的表示方法之后，接下来介绍范例中所使用的获胜表的概念。获胜表是一个三阶的布尔数组，数组的前两个索引值代表棋格位置，第3个数组索引值则是游戏中所有获胜组合的编号。下面先来看看范例中获胜表声明的程序代码。

```
bool ptab[10][10][192];    //玩家的获胜表
bool ctab[10][10][192];    //计算机的获胜表
```

前面已经计算出  $10 \times 10$  大小的棋盘共有 192 种获胜组合，因此上面获胜表数组中第3个索引值的范围是 0~191，其中每个索引值都代表着各个获胜组合的编号。像这样的—个获胜表数组，数组元素所表示的意义就是某一棋格是否位于某个获胜组合之中。

当获胜表数组中的元素值位于某个获胜组合中时，便将其设定为“true”。举个例子，假设棋格“[3][2]”这个位置位于编号为“1”、“99”、“111”的获胜组合中，以玩家的获胜表来说，下面所列这些元素的值将会是“true”。

```
ptab[3][2][1]      = true;
ptab[3][2][99]     = true;
ptab[3][2][111]    = true;
```

此外，在棋局进行时，上面这些玩家获胜表元素为“true”的条件，必须是棋格“[3][2]”为“空”或为“玩家的棋子”，这样才表示玩家有可能在“[3][2]”这个位置上由编号为“1”、“99”、“111”的这些获胜组合中的其中一个获胜。反之，若棋格“[3][2]”被计算机的棋子占用了，玩家便无法在“[3][2]”这个位置上放棋子，因此各个元素的值将会被设定为“false”。

```
ptab[3][2][1]      = false;
ptab[3][2][99]     = false;
ptab[3][2][111]    = false;
```

相反，计算机获胜表中相对元素的值则会是“true”。

```
ctab[3][2][1]      = true;
ctab[3][2][99]     = true;
ctab[3][2][111]    = true;
```

程序利用玩家与计算机两份获胜表记录在棋局进行时双方所下的棋子是否能够在某些获胜组合上达成五子相连，其中的内容也是设计计算机下棋 AI 时用来计算出最佳下子位置的依据。

在棋局开始时，程序会按照每个棋格位置属于哪些获胜组合来设定获胜表数组的初始内容。由于刚开始时棋盘上并没有任何棋子，因此玩家与计算机获胜表内容的初值是一样的。之后随着棋局



的进行, 玩家与计算机在棋盘上下棋子, 两者获胜表数组中的元素值便随着所下棋子的位置以前面所述的方式进行设定。

此外, 程序还利用一个如下的数组来记录玩家与计算机在 192 种获胜组合中各填入了几颗棋子。

```
int win[2][192];
```

上面的这个二维数组中, 第 1 个数组索引值代表玩家或者计算机, 范例中以“0”代表玩家, “1”代表计算机; 第 2 个索引值代表获胜组合的编号。每个数组元素值用来记录玩家或者计算机在各个获胜组合中填入了几颗棋子。

“win”数组的所有元素初值都是“0”, 每当计算机或玩家在棋盘上放置一颗棋子后, 程序会按照棋子的位置去获胜表中找出包含该位置的所有获胜组合, 接着累加玩家或计算机在这些获胜组合中填入的棋子数目。以前面的例子来说, 当玩家在“[3][2]”这个位置上放置了一颗棋子之后, 就等于玩家在包含“[3][2]”这个位置编号为“1”、“99”、“111”的获胜组合中填入了一颗棋子, 因此“win”数组中记录玩家在这些获胜组合中填入棋子总数的元素必须累加“1”, 运行的程序代码如下所示。

```
win[0][1]++;           //玩家在编号“1”的获胜组合中的棋子数目累加“1”
win[0][99]++;          //玩家在编号“99”的获胜组合中的棋子数目累加“1”
win[0][111]++;         //玩家在编号“111”的获胜组合中的棋子数目累加“1”
```

“win”数组的元素值在正常的情况下为 0~6, 表示计算机或玩家在各个获胜组合中填入了多少颗棋子, 当元素值等于“5”或“6”时, 便表示有一方已经完成了某一获胜组合的五子联机而赢得棋局。当玩家或计算机某一方获胜组合中的位置被对方占走, 那么便无法在包含被占走位置的获胜组合上完成五子相连, 这样“win”数组中对应的元素值会直接设定为“7”, 用来表示该方已无法由被占走位置的获胜组合上赢得棋局。

延续上面的例子, 当玩家在“[3][2]”这个位置上下了一颗棋子之后, 除了必须将“win”数组里代表玩家获胜组合中包含位置“[3][2]”的元素值累加“1”外, 还必须将代表计算机获胜组合中包含位置“[3][2]”的元素值设为“7”, 程序代码如下。

```
win[1][1] = 7;         //计算机已无法由编号“1”的获胜组合上赢得棋局
win[1][99] = 7;        //计算机已无法由编号“99”的获胜组合上赢得棋局
win[1][111] = 7;       //计算机已无法由编号“111”的获胜组合上赢得棋局
```

以上所讨论的获胜表与记录计算机和玩家双方在各个获胜组合中填入棋子个数的“win”数组, 是五子棋范例中设计计算机决策 AI 的重要关键。下面将继续说明整个五子棋游戏计算机决策 AI 的设计重点, 也就是关于“棋格获胜分数”的计算方法与运用。

### 3. 计算棋格获胜分数

在五子棋范例中, 计算机决策 AI 的设计概念是在每次下棋子之前先计算所有空白棋格上的获胜分数, 接着再根据获胜分数的高低来决定哪一个空白棋格是最佳的下子位置。在范例中, 获胜分数越高的棋格便表示在这个棋格上下棋子, 将会有较高的获胜几率。

获胜分数的计算规则, 是以包含棋格所在位置的获胜组合数目, 以及目前棋盘上这些获胜组合中已存在的棋子数目的多寡来累加棋格的获胜分数。下面以几个例子来说明获胜分数的计算规则。



## 1) 在可达成的获胜组合上拥有越多棋子的棋格分数值越高

就单一获胜组合而言，棋格的获胜分数是以可达成的获胜组合上已放置的棋子数来设定的。例如在图 5-24 中，假设当某一获胜组合中已放置 2、3 和 4 颗棋子时，其上空棋格的获胜分数分别为“20”、“50”和“1000”。

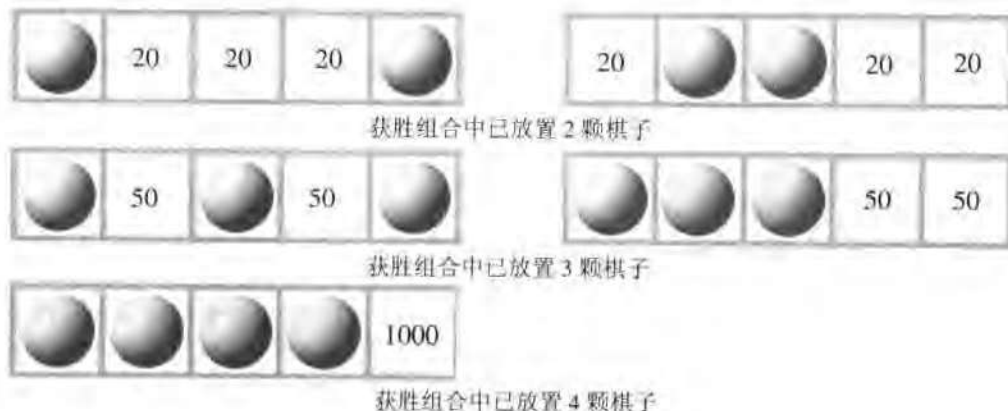


图 5-24

由图中的说明可以理解，当可获胜组合中已放置的棋子数越多，那么在这个获胜组合上的空棋格上下棋子而赢得棋局的机会就越大，所以就设定越高的分数，尤其是当已放置了 4 颗棋子时，必须在第 5 个空棋格上设定绝对高值的分数。

除了上面所介绍的设定棋格分数的基本方法外，还必须考虑当获胜组合上有部分位置已被对手棋格占走而无法达成五子联机的状况时，获胜组合上空棋格的获胜分数会直接设定为“0”。图 5-25 便是当获胜组合中已填入两颗棋子，对于未被对手占走棋格位置及已被对手占走棋格位置的两种情况在获胜分数设定上的比较。

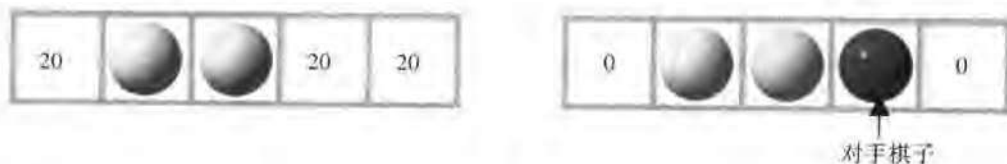


图 5-25

在右边的图例中，当获胜组合中有任一位置被对手的棋子占走时，那么组合上的所有空棋格的获胜分数会直接设定为“0”。这是在单一获胜组合的情况下。事实上由于棋盘上的棋格可能会位于多种获胜组合之中，因而棋格上的获胜分数总会以累加的方式来计算。

对于判断获胜组合中的位置是否被对手的棋子所占走，或者获胜组合中填入了多少颗棋子，则参考前面所提过的获胜表及“win[]”数组中所记录的内容。

## 2) 按照棋格所在位置上可达成联机的获胜组合总数进行分数加总

范例中所使用的 10×10 棋盘上共有 192 种获胜组合，而棋格的获胜分数则是在所有单一获胜组合上分数的加总，当加总后的获胜分数越高，就表示在棋格上下棋子会有较高的获胜几率。下面我们简单地利用一个 5×5 大小的棋盘来解释这样的计算方法，如图 5-26 所示。





	0	1	2	3	4
0			20	25	40
1	10	30	5		5
2	5	20	25	5	0
3	10		5	15	5
4	25	20	0	5	5

图 5-26

图中空白棋格中的数字部分就是棋格加总后的获胜分数，这里仅以每一获胜组合中最多两颗棋子为例进行说明，并且假设当棋格放在已放置了一颗棋子的获胜组合上时分数为“5”，而放在已放置两颗棋子的获胜组合上时分数为“20”。

以盘面上所标示的最佳放棋子的位置“[0][4]”来说，由于它所在的水平与反对角线方向上的获胜组合中都包含了两颗棋子，因此对单一获胜组合来说，该位置的获胜分数都为“20”，而累加的结果便是“40”。接着再考虑图中“[3][0]”这个位置，由于它所在的水平与垂直方向上的获胜组合中各包含了一颗棋子，而单一获胜组合中包含一颗棋子的获胜分数为“5”，因此“[3][0]”这个位置上的获胜分数便为“10”。

按照相同的概念，可以推论出其他棋格上的获胜分数。在游戏进行时的实际计算，还必须判断各个单一获胜组合上是否有某些棋格位置已被对手的棋子占走而无法达成，并按获胜组合中已填入的棋子数来增减获胜分数的比重，最后累加所有单一组合上棋格的获胜分数，便是棋格真正的获胜分数。

#### 4. 计算机的攻击与防守

了解了棋格获胜分数的计算方法，要设计计算机的攻击与防守的决策 AI 就相当简单了。在前面讨论获胜分数的计算时，可以知道所计算出获胜分数越高的棋格就是最有可能使自己的棋子达成联机的位置，也就是最佳的攻击位置，而一旦计算机能够找出分数最高的棋格来下棋子，计算机 AI 就具备攻击的能力了。

那么计算机的防守能力要如何来设计呢？其实道理还是相同的，计算机需要进行防守，也就代表着玩家在棋格上具有较高的获胜分数而有较高达成获胜组合的几率。因此程序中会利用相同的方法计算玩家在空棋格上的获胜分数，接着比较计算机与玩家哪一方具有最高的获胜分数来决定进行攻击或防守。当玩家在某一棋格上拥有最高的获胜分数时，表示玩家在此棋格上具有最大达成联机的可能，此时计算机必须将下一步棋子摆在这一位置上，用来阻止对手达成获胜组合来进行防守，而当计算机在棋格上拥有最高的获胜分数时，则是棋子下在这个最佳攻击位置上来进行攻击。

在看过了整个五子棋游戏中关于计算机决策 AI 的设计原理之后，下面就来看看整个范例的完整程序内容及其说明。

### 》范例 ch5\_4

玩家与计算机对战五子棋策略型 AI 游戏范例。

#### 程序代码：全局变量声明

```

1  //全局变量声明
2  HINSTANCE hInst;
3  HBITMAP chess[2];
4  HDC      hdc,mdc,bufdc;
5  HWND      hWnd;
6  DWORD     tPre,tNow;
7  int       board[10][10];
8  bool      ptab[10][10][192];      //玩家获胜表
9  bool      ctab[10][10][192];      //计算机获胜表
10 int       win[2][192];
11 int       num[2];
12 bool      turn,over;
13 int       winner;
```

#### 程序说明

(1) 第 3 行：声明位图对象数组，两个数组元素“chess[0]”与“chess[1]”分别存储玩家与计算机的棋子图案。

(2) 第 7 行：声明 10×10 的二阶整数数组“board[]”，其数组元素用来存储目前棋盘上所有棋格的状态。元素值有“0”、“1”和“2”，分别用来代表玩家的棋子、计算机的棋子和空格。

(3) 第 11 行：声明整数数组“num[]”，其中的两个元素“num[0]”与“num[1]”分别用来记录目前玩家与计算机所下的棋子个数。记录玩家与计算机所下的棋子个数的目的是要用来判断双方是否是平手。由于范例中使用 10×10 大小的棋盘，因此当玩家与计算机双方所下的棋子个数都等于“50”时，便表示双方是平手。

(4) 第 12 行：声明两个布尔变量“turn”与“over”。“turn”变量用来指示目前由哪一方下棋子，其值为“true”时表示由玩家下棋子，为“false”时则表示由计算机下棋子；“over”变量则是用来指示棋局是否结束，其值为“true”时表示棋局结束。

(5) 第 13 行：声明“winner”变量，用来记录当游戏结束时由哪一方赢得棋局。“winner”的设定值有“0”、“1”与“2”，分别代表玩家、计算机与平手。

#### 程序代码：函数声明

```

1  //函数声明
2  ATOM      MyRegisterClass(HINSTANCE hInstance);
3  BOOL      InitInstance(HINSTANCE, int);
4  LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
5  void      MyPaint(HDC hdc);
6  void      InitGame();
7  void      ComTurn();
```

#### 程序说明

(1) 第 6 行：声明 InitGame()函数，该函数中是游戏开始时各项数据数值的初始操作。

(2) 第 7 行：声明 ComTurn()函数，为轮到计算机下棋子时所调用的函数。此函数会进行获胜

分数的计算工作,以决定计算机的下了位置。

**程序代码: InitInstance()**

```
1  //****初始化函数*****
2  // 棋盘拼接以及调用 InitGame() 函数开始棋局
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {
5      HBITMAP tile,bmp;
6      int rowNum,colNum;
7      int i,x,y;
8
9      hInst = hInstance;
10     hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
11         CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
12
13     if (!hWnd)
14     {
15         return FALSE;
16     }
17
18     MoveWindow(hWnd,10,10,480,520,true);
19     ShowWindow(hWnd, nCmdShow);
20     UpdateWindow(hWnd);
21
22     hdc = GetDC(hWnd);
23     mdc = CreateCompatibleDC(hdc);
24     bufdc = CreateCompatibleDC(hdc);
25
26     bmp = CreateCompatibleBitmap(hdc,450,450);
27     SelectObject(mdc,bmp);
28
29     tile = (HBITMAP)LoadImage(NULL,"tile.bmp",IMAGE_BITMAP,45,45,LR_LOADFROMFILE);
30     chess[0] = (HBITMAP)LoadImage(NULL,"chess0.bmp",IMAGE_BITMAP,38,38,
        LR_LOADFROMFILE);
31     chess[1] = (HBITMAP)LoadImage(NULL,"chess1.bmp",IMAGE_BITMAP,38,38,
        LR_LOADFROMFILE);
32
33     for (i=0;i<100;i++)
34     {
35         rowNum = i / 10;
36         colNum = i % 10;
37         x = colNum * 45;
38         y = rowNum * 45;
39
40         SelectObject(bufdc,tile);
41         BitBlt(mdc,x,y,45,45,bufdc,0,0,SRCCOPY);
42     }
43
44     InitGame();
```

```

45  MyPaint(hdc);
46
47  return TRUE;
48  }

```

## 程序说明

InitInstance()函数中进行了所有图案的加载操作。其中第33~42行程序代码以拼接的方法先在mdc中产生游戏中所用的棋盘,之后第44行程序代码则调用InitGame()函数来进行棋局开始的各项初始操作。

## 程序代码: InitGame()

```

1  //****棋局初始函数*****
2  // 1.设定棋盘初始状态及获胜表内容
3  // 2.决定先下子的一方
4  void InitGame()
5  {
6      int i,j,k;
7      int count=0;
8
9      over = false;
10     num[0] = num[1] = 0;
11
12     //设定玩家与计算机在各个获胜组合中的棋子数
13     for(i=0;i<192;i++)
14     {
15         win[0][i] = 0;
16         win[1][i] = 0;
17     }
18
19     //初始化棋盘状态
20     for(i=0;i<10;i++)
21         for(j=0;j<10;j++)
22             board[i][j] = 2;
23
24     //设定水平方向的获胜组合
25     for(i=0;i<10;i++)
26         for(j=0;j<6;j++)
27         {
28             for(k=0;k<5;k++)
29             {
30                 ptab[i][j+k][count] = true;
31                 ctab[i][j+k][count] = true;
32             }
33             count++;
34         }
35
36     //设定垂直方向的获胜组合
37     for(i=0;i<10;i++)
38         for(j=0;j<6;j++)

```

```
39     {
40         for(k=0;k<5;k++)
41         {
42             ptab[j+k][i][count] = true;
43             ctab[j+k][i][count] = true;
44         }
45         count++;
46     }
47
48     //设定正对角线方向的获胜组合
49     for(i=0;i<6;i++)
50         for(j=0;j<6;j++)
51         {
52             for(k=0;k<5;k++)
53             {
54                 ptab[j+k][i+k][count] = true;
55                 ctab[j+k][i+k][count] = true;
56             }
57             count++;
58         }
59
60     //设定反对角线方向的获胜组合
61     for(i=0;i<6;i++)
62         for(j=9;j>=4;j--)
63         {
64             for(k=0;k<5;k++)
65             {
66                 ptab[j-k][i+k][count] = true;
67                 ctab[j-k][i+k][count] = true;
68             }
69             count++;
70         }
71
72     //随机数决定由哪一方先下棋子
73     srand(GetTickCount());
74     if(rand()%2 == 0)
75         turn = true;
76     else
77         turn = false;
78 }
```

#### 程序说明

- (1) 第7行：声明变量“count”，用来在后面程序设定获胜表中的内容时作为获胜组合的编号。
- (2) 第13~17行：以循环将玩家与计算机在各个获胜组合中填入的棋子个数设为“0”。
- (3) 第20~22行：设定棋盘上所有的棋格都为空棋格（数值为2）。
- (4) 第24~69行：按照顺序求出水平、垂直、正对角线和反对角线方向上各个获胜组合所包含棋格的位置，并设定玩家与计算机获胜表中对应的元素值为“true”。
- (5) 第25~70行：这里以水平方向的获胜组合为例来说明设定获胜表中内容的方法。这段程

序代码中，第 1 次循环运行时所要设定的获胜组合如图 5-27 所示。

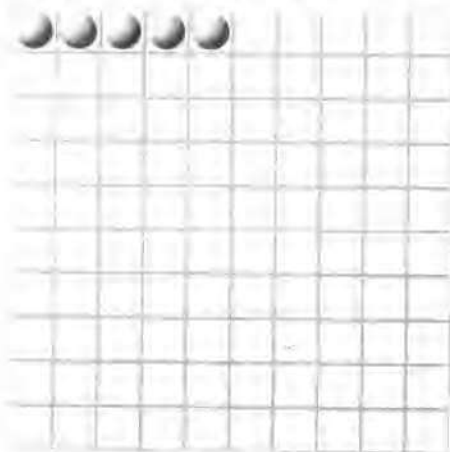


图 5-27

图中的排列方式为 192 种获胜方法中的第 1 种，第 30、31 行程序代码中“count”的值一开始为“0”，也就是这个获胜组合的编号。接着再观察“i”、“j”、“k”值的变化，第 2 层循环会递增“j”的值，也就是格子的位置会在水平方向上由左向右移动，第 1 层循环则是在第 2 层循环运行完以后往下一列移动，“k”与“j”值相加则是代表某一种获胜组合中的 5 颗棋子在棋盘上水平方向上的索引值，例如，图 5-27 第 1 种获胜组合中的五颗棋子位置为：

```
[0][0], [0][1], [0][2], [0][3], [0][4]
```

因此，这一个获胜组合便会使得玩家与计算机获胜表中设定的初始内容如下。

```
ptab[0][0][0] = true;
ptab[0][1][0] = true;
ptab[0][2][0] = true;
ptab[0][3][0] = true;
ptab[0][4][0] = true;
```

第 33 行程序代码在初始化一种获胜组合后便累加“count”的值（获胜组合编号加“1”），继续求出放在下一个获胜组合中的棋格位置，并设定获胜表中的内容如下。

```
ptable[0][1][1] = true;
ptable[0][2][1] = true;
ptable[0][3][1] = true;
ptable[0][4][1] = true;
ptable[0][5][1] = true;
```

按照上面这样的方法，便可一一推算出棋盘各方向上的所有获胜组合位置并设定玩家与计算机获胜表的初始内容。

(6) 第 73~77 行：以随机数决定是玩家还是计算机先下，当“turn”为“true”时为玩家先下，“turn”为“false”时则为计算机先下。

程序代码：WndProc()

```
1  //****消息处理函数*****
2  // 1. 设定按下【F1】键重新开始游戏
```

```

3 // 2.处理玩家单击鼠标左键下棋子的动作
4 LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
    lParam)
5 {
6     int x,y,m,n,i;
7
8     switch (message)
9     {
10         case WM_KEYDOWN:                //按下按键消息
11             switch (wParam)
12             {
13                 case VK_ESCAPE:          //按下【Esc】键
14                     PostQuitMessage( 0 );
15                     break;
16                 case VK_F1:              //按下【F1】键
17                     InitGame();
18                     break;
19             }
20         case WM_LBUTTONDOWN:            //单击鼠标左键消息
21             if(!over)
22                 if(turn)
23                 {
24                     x = LOWORD(lParam); //取得鼠标 X 坐标
25                     y = HIWORD(lParam); //取得鼠标 Y 坐标
26
27                     if(x > 10 && x < 460 && y > 10 && y < 460)
28                     {
29                         m = (int)floor((double)((x-10)/45));
30                         n = (int)floor((double)((y-10)/45));
31
32                         if(board[m][n] == 2)
33                         {
34                             board[m][n] = 0; //设定为玩家的棋子
35                             num[0]++;
36
37                             if(num[0] == 50 && num[1] == 50)
38                             {
39                                 winner = 2;        //平手
40                                 over = true;
41                             }
42                             else
43                                 for(i=0;i<192;i++)
44                                 {
45                                     if(ptab[m][n][i])
46                                     {
47                                         win[0][i]++;
48                                         ctab[m][n][i] = false;
49                                         win[1][i] = 7;
50

```



```

51                                     if(win[0][i] == 5)
52                                     {
53                                         winner = 0;
54                                         over = true;
55                                     }
56                                     }
57                                     }
58                                     turn = false;    //换由计算机下
59                                     }
60                                     }
61                                     }
62                                     break;
63     case WM_DESTROY:                                //窗口结束消息
64         DeleteDC(mdc);
65         DeleteDC(bufdc);
66         DeleteObject(chess[0]);
67         DeleteObject(chess[1]);
68
69         ReleaseDC(hWnd, hdc);
70
71         PostQuitMessage(0);
72         break;
73     default:                                         //其他消息
74         return DefWindowProc(hWnd, message, wParam, lParam);
75     }
76     return 0;
77 }

```

## 程序说明

消息处理函数将处理玩家单击鼠标左键下棋子的动作，并提供按下【F1】键开始新棋局的功能。

(1) 第 16~19 行：当按下的按键为【F1】键时，调用 InitGame()函数重新初始化各项数据，开始新棋局。

(2) 第 20~22 行：当玩家单击了鼠标左键时，判断目前游戏是否结束，若未结束则再判断是否轮到玩家下棋子。

(3) 第 24~27 行：当轮到玩家下棋子时，则取得单击鼠标时光标的所在坐标，并判断此坐标是否落在棋盘内。

(4) 第 29~30 行：若单击鼠标时光标所在坐标落在棋盘内，则计算该坐标所在的棋格位置索引值“m”和“n”，其中 floor()函数为无条件舍去尾数函数。

(5) 第 32~35 行：判断玩家单击的棋格位置是否为空棋格。若为空棋格，则将该棋格的状态设为玩家的棋子，并累加目前玩家所下的棋子总数。

(6) 第 37~41 行：在玩家下了之后，判断目前玩家与计算机的棋子总数是否都等于 50。若是，则表示此棋局已结束且双方平手，将“winner”的值设为“2”并将“over”的值设为“true”。

(7) 第 43~57 行：若玩家此次的下子操作未使双方平手，便以循环的方式对照玩家获胜表中的内容。当判断棋子的位置位于某获胜组合中时，第 47 行程序代码便累加玩家在该获胜组合中的棋子总数。接着第 48 行程序代码设定计算机获胜表中相同位置上的获胜可能为“false”，第 49 行程序代码设定计算机在该获胜组合中填入的棋子总数为“7”（“7”代表无法由此获胜组合上获胜）。

(8) 第51~55行: 判断此次的下子操作是否在可获胜组合中完成五子相连。若是, 则代表玩家赢得棋局。将“over”设为“true”并将“winner”设为“0”。

(9) 第58行: 在玩家下完棋子后, 设定“turn”为“false”, 换由计算机下子。

程序代码: ComTurn()

```

1  //****计算机下子函数*****
2  // 1.计算获胜分数
3  // 2.选择最佳位置进行下子操作
4  void ComTurn()
5  {
6      int grades[2][10][10];
7      int m,n,i,max=0;
8      int u,v;
9
10     for(m=0;m<10;m++)
11         for(n=0;n<10;n++)
12             {
13                 grades[0][m][n] = 0;
14                 grades[1][m][n] = 0;
15
16                 if(board[m][n] == 2)
17                     {
18                         for(i=0;i<192;i++)
19                             {
20                                 //计算玩家在空棋格上的获胜分数
21                                 if(ptab[m][n][i] && win[0][i] != 7)
22                                     {
23                                         switch(win[0][i])
24                                         {
25                                             case 0:
26                                                 grades[0][m][n] += 1;
27                                                 break;
28                                             case 1:
29                                                 grades[0][m][n] += 200;
30                                                 break;
31                                             case 2:
32                                                 grades[0][m][n] += 400;
33                                                 break;
34                                             case 3:
35                                                 grades[0][m][n] += 2000;
36                                                 break;
37                                             case 4:
38                                                 grades[0][m][n] += 10000;
39                                                 break;
40                                         }
41                                     }
42
43                                 //计算计算机在空棋格上的获胜分数
44                                 if(ctab[m][n][i] && win[1][i] != 7)

```

```

45         {
46             switch(win[1][i])
47             {
48                 case 0:
49                     grades[1][m][n] += 1;
50                     break;
51                 case 1:
52                     grades[1][m][n] += 220;
53                     break;
54                 case 2:
55                     grades[1][m][n] += 420;
56                     break;
57                 case 3:
58                     grades[1][m][n] += 2100;
59                     break;
60                 case 4:
61                     grades[1][m][n] += 20000;
62                     break;
63             }
64         }
65     }
66
67     if(max == 0)
68     {
69         u = m;
70         v = n;
71     }
72
73     if(grades[0][m][n] > max)
74     {
75         max = grades[0][m][n];
76         u = m;
77         v = n;
78     }
79     else if(grades[0][m][n] == max)
80     {
81         if(grades[1][m][n] > grades[1][u][v])
82         {
83             u = m;
84             v = n;
85         }
86     }
87
88     if(grades[1][m][n] > max)
89     {
90         max = grades[1][m][n];
91         u = m;
92         v = n;
93     }

```

```

94         else if(grades[1][m][n] == max)
95         {
96             if(grades[0][m][n] > grades[0][u][v])
97             {
98                 u = m;
99                 v = n;
100             }
101         }
102     }
103 }
104
105 board[u][v] = 1;    //设定为计算机的棋子
106 num[1]++;
107
108 if(num[0] == 50 && num[1] == 50)
109 {
110     winner = 2;      //平手
111     over = true;
112 }
113 else
114     for(i=0;i<192;i++)
115     {
116         if(ctab[u][v][i])
117         {
118             win[1][i]++;
119             ptab[u][v][i] = false;
120             win[0][i] = 7;
121
122             if(win[1][i] == 5)
123             {
124                 winner = 1;
125                 over = true;
126             }
127         }
128     }
129 turn = true;        //换由玩家下
130 }

```

#### 程序说明

在 ComTurn()函数中, 计算机会计算玩家与计算机在每一个空棋格上的获胜分数, 并将棋子下在两者中获胜分数最高的棋格上。

(1) 第 6 行: 声明三阶数组 “grades”。数组中的各个元素用来记录玩家及计算机在每个棋格上的获胜分数。数组的第 1 个索引值为 “0” 代表玩家, 为 “1” 则代表计算机。

(2) 第 13~16 行: 在以循环计算玩家与计算机在棋盘上各个棋格的获胜分数时, 先将棋格的分数设为 “0”。第 16 行程序代码则是当棋格为空格时才进行分数的计算, 否则表示该格子已被棋子占住, 不需进行分数的计算, 分数为 “0”, 因为根本不可能再把棋子摆到已有棋子的位置上。

(3) 第 18~65 行: 以循环判断每个棋格是否位于玩家或计算机的获胜组合中及双方在棋格上

是否有取得获胜的机会，并进行获胜分数的计算。

(4) 第 21~41 行：计算玩家在空棋格上的获胜分数。在第 21 行程序代码判断的条件式中，必须当棋格位于玩家的获胜组合中（`ptab[m][n][i]==true`）且玩家在该获胜组合上有可能达成时（`win[0][i]!=7`）才进行下面的分数累加操作，否则该棋格在第 *i* 种单一获胜组合上的获胜分数即为前面所设的初值“0”。

(5) 第 23~40 行：在计算玩家的获胜分数时，根据目前玩家在第 *i* 种获胜组合上所填入的棋子个数进行分数累加操作。当获胜组合中的棋子个数为“0”、“1”、“2”、“3”和“4”时，其加权分数按照顺序递增，分别为“1”、“200”、“400”、“2000”和“10000”。

(6) 第 44~64 行：按照前面计算玩家在棋格获胜分数的方法，计算计算机在棋格上的获胜分数。当计算机分数的加权值高于玩家分数的加权值时，可以让计算机在与玩家势均的情况下选择将棋子下在可使自己达成获胜组合的位置上进行攻击。

(7) 第 67~71 行：“max”变量是用来记录玩家与计算机两者间棋格分数的最大值，拥有最大棋格分数的位置就是计算机下棋子的位置。如果最大分数为玩家的获胜分数，那么计算机进行防守，而如果最大分数是计算机的获胜分数，那么计算机进行攻击。具有最大获胜分数的棋格位置记录在变量“u”与变量“v”中。在大部分的情况下，“max”、“u”、“v”的值会随着后面的程序中找到 `grades` 数组中的最大值而改变，当玩家与计算机即将平手而棋盘上仍有空棋格时，`grades` 数组中的值会全部为“0”，因为没有任何一方可以获胜，这段程序代码便会记录一个空棋格“[u][v]”，成为计算机下棋子的位置。

(8) 第 73~78 行：每当计算完一棋格上玩家与计算机的获胜分数后，程序会先判断这一棋格上的玩家获胜分数是否大于之前所记录的最大获胜分数“max”，若是，就以这个棋格上的玩家获胜分数来更新“max”的值，同时改变计算机所要下棋子的位置“u”和“v”。

(9) 已知一个棋格位置是较佳的位置，因此第 81 行判断条件式中则是去比较棋格与“[u][v]”位置上计算机获胜分数的高低，当棋格上计算机的获胜分数高于上次所决定的“[u][v]”这个位置上的分数时，便以棋格的位置取代“u”和“v”的值。单就玩家的获胜分数来看，第 73~86 行程序代码所决定出的“[u][v]”会是最佳防守位置中的最佳的攻击位置。下面以图 5-28 的流程图来说明以上计算机选择最佳下子位置的决策过程。

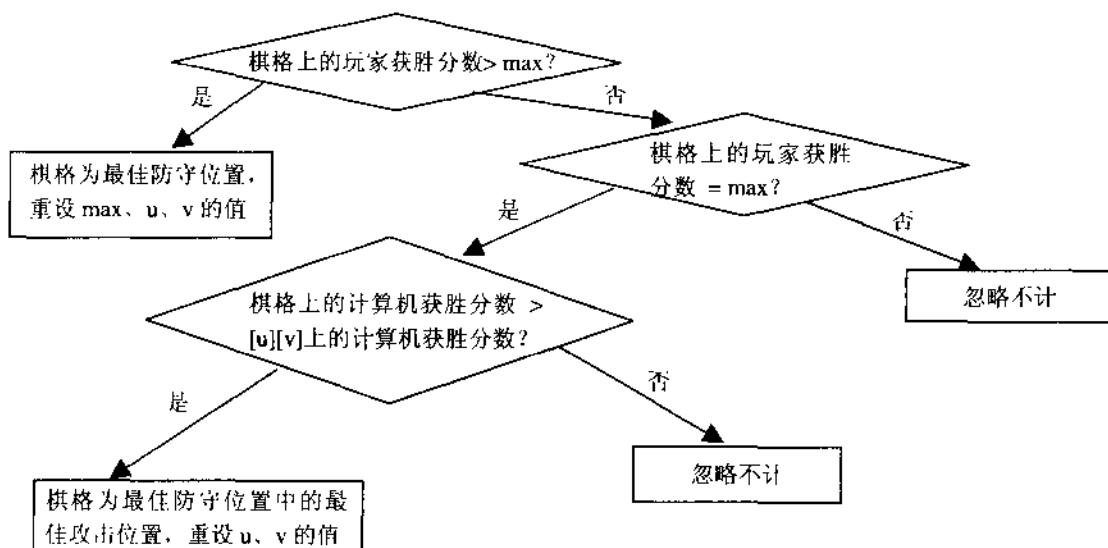


图 5-28

(10) 第 88~101 行: 在判断完棋格上的玩家获胜分数之后, 接着再按照相同的方法判断计算机的获胜分数, 并根据是否为最大获胜分数“4”来重设“max”、“u”和“v”的值。

(11) 第 105~106 行: 经过前面的循环顺序, 判断所有棋格上的获胜分数以决定最佳的下子位置之后, 最后的结果便会存储在“u”和“v”中。第 105 行程序代码便将“[u][v]”这个位置设定为计算机的棋子, 接着再累加目前计算机所下棋子的总数。

(12) 第 108~112 行: 在计算机下棋子之后, 判断双方是否是平手。

(13) 第 114~128 行: 以循环判断计算机下棋子的位置落在哪些获胜组合当中, 并累加获胜组合中填入的棋子总数, 重设玩家获胜表及 win[] 数组中玩家不可能达成五子相连的获胜组合。最后再判断计算机是否完成五子相连而赢得棋局。

(14) 第 129 行: 将“turn”的值设为“true”, 换由玩家下。

#### 程序代码: MyPaint()

```

1  //****自定义绘图函数*****
2  // 窗口贴图及显示消息
3  void MyPaint(HDC hdc)
4  {
5      int m,n;
6      char *str;
7
8      if(over)
9      {
10         switch(winner)
11         {
12             case 0:
13                 str = "您赢了！按下【F1】键可重新进行游戏..";
14                 break;
15             case 1:
16                 str = "计算机赢了！按下【F1】键可重新进行游戏..";
17                 break;
18             case 2:
19                 str = "不分胜负！按下【F1】键可重新进行游戏..";
20                 break;
21         }
22         TextOut(hdc,10,470,str,strlen(str));
23     }
24     else if(!turn) //计算机下子
25     {
26         str = "计算机思考中...";
27         TextOut(hdc,10,470,str,strlen(str));
28         ComTurn();
29     }
30     else
31     {

```

```

32     str = "该您下了...";
33     TextOut(hdc,10,470,str,strlen(str));
34 }
35
36 for(m=0;m<10;m++)
37     for(n=0;n<10;n++)
38     {
39         if(board[m][n] == 0)           //贴上玩家棋子
40         {
41             SelectObject(bufdc,chess[0]);
42             BitBlt(mdc,m*45+3,n*45+3,38,38,bufdc,0,0,SRCCOPY);
43         }
44         else if(board[m][n] == 1)       //贴上计算机棋子
45         {
46             SelectObject(bufdc,chess[1]);
47             BitBlt(mdc,m*45+3,n*45+3,38,38,bufdc,0,0,SRCCOPY);
48         }
49         else                             //贴上空格
50         {
51             SelectObject(bufdc,chess[1]);
52             BitBlt(mdc,m*45+3,n*45+3,38,38,bufdc,0,0,WHITENESS);
53         }
54     }
55
56 BitBlt(hdc,10,10,450,450,mdc,0,0,SRCCOPY);
57
58 tPre = GetTickCount();
59 }

```

## 程序说明

(1) 第 8~23 行：判断游戏是否已经结束 (over=true)。若游戏结束，则按“winner”的值输出棋局胜负结果的字符串消息。

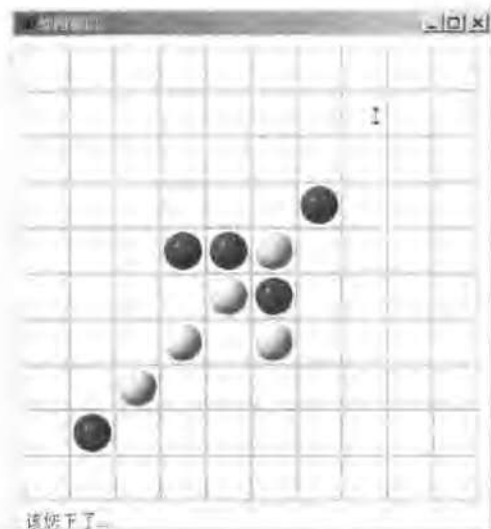
(2) 第 24~29 行：当“turn”值为“false”时，显示“计算机思考中”的消息并调用 ComTurn() 函数进行计算机下子的操作。

(3) 第 30~34 行：当“turn”值为“true”时，显示“由玩家下子”的消息。

(4) 第 36~54 行：以循环顺序判断目前棋盘上每个棋格的状态“board[m][n]”，在棋盘上贴上对应的棋子或空格图案。

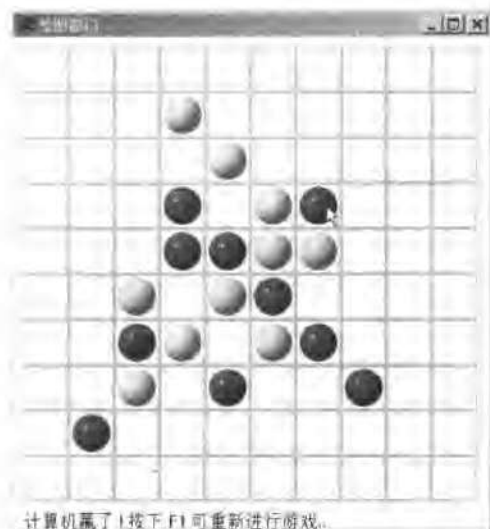
## 运行结果

程序运行结果如图 5-29 和图 5-30 所示。



单击鼠标左键在棋盘上下棋子

图 5-29



被计算机打败了

图 5-30

看过了本章中关于游戏人工智能的介绍，已经能够利用基本的 AI 原理来设计出游戏的骨干并赋予其趣味性了。在开发游戏的过程中，只要能够将上述的原理灵活运用并发挥自己的巧思，便能创造出令人满意的游戏 AI。

## 课后重点整理

- 人工智能 AI (Artificial Intelligence)，主要目的是让计算机可以遵循某些法则来仿真出类似人类般的思考与预测能力，并结合计算机具有快速数学运算能力的优点，创造出计算机在各方面的有效应用。
- 类神经网络：以多个联结处理器负责不同单元的处理，仿真人类大脑思考与学习能力的人工智能理论。
- 基因算法：利用仿真自然界适者生存的进化原理，对于问题产生最佳化解决方案的人工智能理论。
- 模糊逻辑：以一种判断推理 (if-else) 的方式来产生最佳猜测的决定，有别于一般以数学运算为基础的人工智能理论。
- 行为型游戏 AI 主要是关于计算机角色本身的判断思考，而后产生对应行为的 AI。在设计行为型的游戏 AI 时，通常会利用到一连串的“if-else”判断、数学运算，或者一些数据结构的概念。
- 在搜寻迷宫出口的过程中，计算机必须对于接下来该往哪个方向移动做思考判断。
- 在搜寻迷宫出口游戏中，除了判断下一时刻移动的目的之外，还必须记录计算机角色走过的迷宫的路径。
- 在搜寻迷宫出口范例中，用到了行为型 AI 及地图拼接的原理，并利用堆栈 (stack) 数据结构来存储搜寻迷宫时所经过的每个迷宫方格的编号。
- 堆栈的删除操作是直接删除末端指针所指的元素，再将末端指针指向堆栈的最后一个元素。



- 运用策略型 AI 最常见的游戏便是棋盘式游戏。在这种类型的游戏中，通常计算机必须按照目前的状况来判断所有可走的棋步与可能的获胜状况，并在每一回合计算棋盘上计算机或者玩家的可能获胜几率，以决定一个最佳的走法。

## 课后练习

1. 请简述追逐移动的设计原理。
2. 说明什么是模式移动。它和单一的移动模式的主要差异在哪里？
3. 试简述类神经网络、基因算法和模糊逻辑。
4. 追逐移动和躲避移动在设计原理上相当接近，试问两者之间最大的差异点是什么？
5. 搜寻迷宫出口是利用什么数据结构来存储搜寻迷宫时所经过的每个迷宫的方格编号的？
6. 试简述迷宫搜寻概念的算法。
7. 试以图标说明在策略型游戏的五子棋棋局中，一个  $10 \times 10$  大小的五子棋盘的获胜组合有几种？
8. 请问五子棋中计算机的攻击与防守的决策 AI 的主要准则是什么？

## 第 6 章 游戏物理现象设计原理

### 6.1 物理运动

将真实世界中的物理现象呈现于游戏中，对于游戏设计来说是相当重要的一门课题，例如物体移动、碰撞或者物体爆破然后碎片飞散等，都属于物理现象。程序中要制作这些真实世界中的物理现象的效果，需要用到高中所学的物理概念与数学计算。

这一节里，将从各种物理运动开始，说明游戏中物理效果的运用时机与设计方式。

#### 6.1.1 匀速运动

物体会移动，那这个物体一定具有“速度”，速度是物体在各个方向上“速度分量”的合成。以一个在 2D 平面上移动的物体而言，假设它的移动速度为  $V$ ， $X$  轴方向上的速度分量为  $V_x$ ， $Y$  轴方向上的速度分量为  $V_y$ ，那么  $V$  与  $V_x$ 、 $V_y$  间的关系如图 6-1 所示。

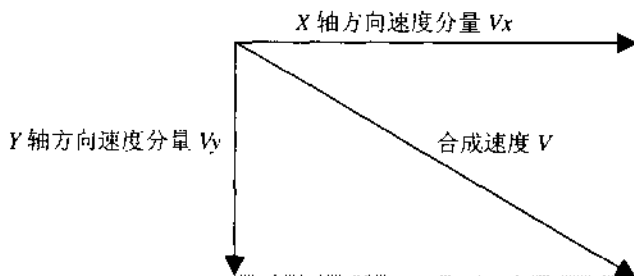


图 6-1

匀速运动的原理就是指物体在每一个时刻的速度都是相同的，即  $V_x$  与  $V_y$  都保持不变。

在设计 2D 平面上物体的匀速运动时，在每次画面更新时，利用物体速度分量  $V_x$  与  $V_y$  的值来计算下次物体出现的位置，产生物体移动的效果。下面将这样的原理以简单的计算公式表示为：

下次  $X$  轴位置 = 现在  $X$  轴位置 +  $X$  轴上速度分量

下次  $Y$  轴位置 = 现在  $Y$  轴位置 +  $Y$  轴上速度分量

看过了关于等速度运动的原理与程序中设计的方法之后，下面就来制作一个物体在 2D 平面上移动的范例程序。

#### » 范例 ch6\_1

设计小球在显示窗口中进行等速度运动，当碰到窗口边缘时则会反弹回来并以反方向运动的程序。

## 程序代码：全局变量声明

```

1  //全局变量声明
2  HINSTANCE  hInst;
3  HBITMAP  bg,ball;
4  HDC      hdc,mdc,bufdc;
5  HWND      hWnd;
6  DWORD     tPre,tNow,tCheck;
7  RECT      rect;
8  int       x=50,y=50,vx=20,vy=20;
  
```

## 程序说明

(1) 第 7 行：用来存储内部窗口区域的矩形结构。

(2) 第 8 行：变量“x”和“y”为窗口中小球的显示（贴图）坐标，“vx”和“vy”则是小球在 X 轴与 Y 轴上运动的速度分量。

## 程序代码：InitInstance()

```

1  //****初始化函数*****
2  // 加载位图并取得内部窗口区域
3  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
4  {
5  HBITMAP bmp;
6  hInst = hInstance;
7
8  hWnd = CreateWindow("canvas", "绘图窗口", WS_OVERLAPPEDWINDOW,
9      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
10
11  if (!hWnd)
12  {
13      return FALSE;
14  }
15
16  MoveWindow(hWnd,10,10,600,450,true);
17  ShowWindow(hWnd, nCmdShow);
18  UpdateWindow(hWnd);
19
20  hdc = GetDC(hWnd);
21  mdc = CreateCompatibleDC(hdc);
22  bufdc = CreateCompatibleDC(hdc);
23  bmp = CreateCompatibleBitmap(hdc,640,480);
24
25  SelectObject(mdc,bmp);
26
27  bg = (HBITMAP)LoadImage(NULL,"bg.bmp", IMAGE_BITMAP,640,480,LR_LOADFROMFILE);
28  ball = (HBITMAP)LoadImage(NULL,"ball.bmp", IMAGE_BITMAP,52,26,LR_LOADFROMFILE);
29
30  GetClientRect(hWnd,&rect);          //取得内部窗口区域大小
31
32  MyPaint(hdc);
  
```

```

33
34 return TRUE;
35 }

```

#### 程序说明

(1) 第 27~28 行: 分别加载背景图与小球图到“bg”和“ball”位图中。此范例中使用的小球图案大小为  $26 \times 26$ , 位图的尺寸与内容如图 6-2 所示。

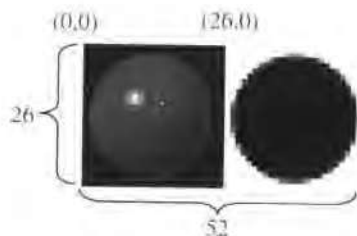


图 6-2

(2) 第 30 行: 调用 GetClientRect() 函数取得内部窗口区域的大小并存储于“rect”中。利用“rect”中存储的内容便可判断出小球移动是否已碰到窗口边缘。

#### 程序代码: MyPaint()

```

1  //*****自定义绘图函数*****
2  // 1. 窗口贴图
3  // 2. 计算小球贴图坐标并判断小球是否碰到窗口边缘
4  void MyPaint(HDC hdc)
5  {
6      SelectObject(bufdc, bg);
7      BitBlt(mdc, 0, 0, 640, 480, bufdc, 0, 0, SRCCOPY);
8
9      SelectObject(bufdc, ball);
10     BitBlt(mdc, x, y, 26, 26, bufdc, 26, 0, SRCAND);
11     BitBlt(mdc, x, y, 26, 26, bufdc, 0, 0, SRCPAINT);
12
13     BitBlt(hdc, 0, 0, 640, 480, mdc, 0, 0, SRCCOPY);
14
15     //计算 X 轴方向贴图坐标与速度
16     x += vx;
17     if(x <= 0)
18     {
19         x = 0;
20         vx = -vx;
21     }
22     else if(x >= rect.right-26)
23     {
24         x = rect.right - 36;
25         vx = -vx;
26     }
27
28     //计算 Y 轴方向贴图坐标与速度

```

```

29  y += vy;
30  if(y<=0)
31  {
32      y = 0;
33      vy = -vy;
34  }
35  else if(y >= rect.bottom-26)
36  {
37      y = rect.bottom - 26;
38      vy = -vy;
39  }
40
41  tPre = GetTickCount(); //记录此次绘图时间
42  }

```

### 程序说明

MyPaint()函数中进行的操作是以前次计算出的贴图坐标先进行窗口透明贴图,之后再计算下一次的贴图坐标。

(1) 第6~13行:窗口贴图,其中9~11行程序代码根据变量“x”与“y”的值进行小球的透明贴图。

(2) 第16行:按照目前小球X轴方向上的速度计算下次X轴上的贴图坐标“x”。

(3) 第17~26行:根据“x”的值判断小球移动是否超出窗口的左缘( $x \leq 0$ )或右缘( $x \geq \text{rect.right}-26$ )。若超出窗口区域,则将“x”位置重新设定,使得X轴上的贴图位置刚好齐切窗口,如图6-3和图6-4所示。

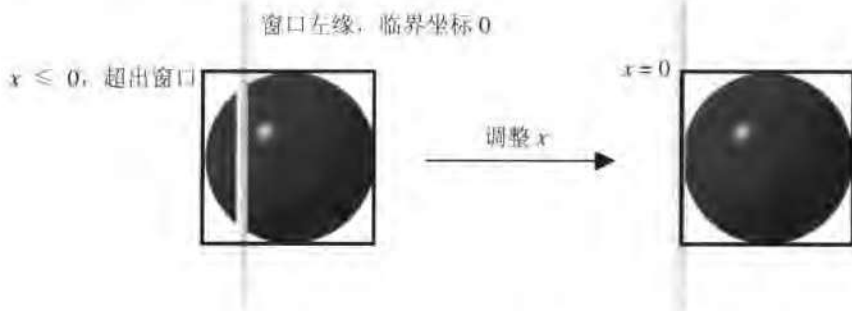


图 6-3



图 6-4

当小球下次在  $X$  轴方向上的贴图坐标碰到窗口的左缘或右缘时,除了重新设定  $X$  轴方向上的贴图坐标值外,同时将  $X$  轴方向上的速度分量设为相反值 ( $vx = -vx$ ),这样小球便会以相反的方向进行移动,产生反弹的效果。

(4) 第 29~39 行:按相同的方法设定小球下次在  $Y$  轴方向上的贴图坐标“ $y$ ”和速度分量“ $vy$ ”。

#### 运行结果

程序运行结果如图 6-5 所示。

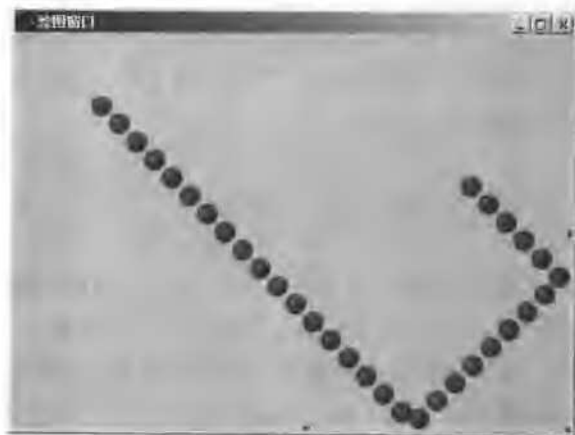


图 6-5

在上面的这个范例中,当小球碰到窗口边缘时会反弹并以相反的方向移动。这是为了让小球能反复在窗口中运动而设计了这样的功能。实际上反弹的动作并不是匀速运动,因为匀速运动必须是速度与方向都为固定,而从小球反弹的物理原理来说,等于是给了小球一个加速度,才使得小球向相反的方向移动。

### 6.1.2 加速度运动

凡是物体移动时其运动的速度或方向会随着时间而改变,那么该物体的运动便是加速度运动。加速度与速度的关系如下:

$$V = V_0 + At$$

公式中,“ $A$ ”表示每一时间间隔加速度的量;“ $t$ ”表示物体运动从开始到要计算时间点为止所经过的时间间隔;“ $V_0$ ”为物体原来所具有的速度;而“ $V$ ”则是由以上公式所计算出某一时间点对象的运动速度。

作用于物体上的加速度同样是各个方向上“加速度分量”的合成,加速度作用于物体上时会根据上面的公式影响物体原有的移动速度。在 2D 平面上运动的物体,根据上面的公式,考虑  $X$  轴、 $Y$  轴上加速度分量对于速度分量的改变,那么其下一刻(前一刻与下一刻时间间隔  $t=1$ )  $X$  轴、 $Y$  轴上的速度分量“ $V_{x1}$ ”与“ $V_{y1}$ ”的计算公式如下:

$$V_{x1} = V_{x0} + A_x$$

$$V_{y1} = V_{y0} + A_y$$

公式中“ $V_{x0}$ ”与“ $V_{y0}$ ”为物体前一刻  $X$  轴、 $Y$  轴上的运动速度,“ $A_x$ ”与“ $A_y$ ”为  $X$  轴、 $Y$  轴上的加速度。

在求出物体下一刻的移动速度后，便可依此再去推算出加入加速度后，物体下一刻的所在位置公式为：

$$Sx1 = Sx0 + Vx1$$

$$Sy1 = Sy0 + Vy1$$

公式中“ $Sx0$ ”与“ $Sy0$ ”分别表示对象前一刻  $X$  轴、 $Y$  轴的坐标位置，“ $Vx1$ ”与“ $Vy1$ ”是加入加速度后下一时刻物体的移动速度，这样求出的“ $Sx1$ ”与“ $Sy1$ ”便是下一刻物体的位置。

前面所介绍的这些和加速度运动相关的计算公式，在设计 2D 平面物体的加速度运动时会用到。只要是物体的移动速度或者方向发生了变化，都是因为受到了加速度的影响。真实世界中的加速度类型可以说是千变万化，如空气阻力、摩擦力、重力等。接下来我们将继续来讨论如何在程序中加入各种类型的加速度，以改变物体的移动状态。

## 6.1.3 重力

牛顿领悟了苹果会从树上落下的道理，是因为受到了地心引力的影响，而地心引力也就是重力，是一个在垂直方向（ $Y$  轴方向）、值大约为  $9.8m/s$ 、方向向下的加速度。

既然重力是一种加速度，那么物体从高处落下，其运动速度与位置坐标的计算，同样可以运用前一小节中所讨论过的物体加速度运动的原理，不过由于重力对于物体运动的影响仅在  $Y$  轴方向，因此不会影响物体在  $X$  轴方向上的速度分量。下面的范例将以物体加速度运动的计算公式与重力的概念，设计小球下坠与弹跳的效果。

### » 范例 ch6\_2

给予小球一个  $X$  轴方向上向右移动的初速度，设计小球由高处下坠及触地反弹的效果，此范例中忽略了小球落下时的空气阻力及触地时的摩擦力。

#### 程序代码：全局变量声明

```
1 //全局变量声明
2 HINSTANCE hInst;
3 HBITMAP bg,ball;
4 HDC hdc,mdc,bufdc;
5 HWND hWnd;
6 DWORD tPre,tNow,tCheck;
7 RECT rect;
8 int x=0,y=100,vx=5,vy=0,gy=1;
```

#### 程序说明

第 8 行：声明小球运动时所会用到的相关变量，如表 6-1 所示。

表 6-1

变 量 名 称	说 明
$x$	小球 $X$ 轴方向上的坐标
$y$	小球 $Y$ 轴方向上的坐标
$vx$	$X$ 轴方向上的速度分量，初值为“5”
$vy$	$Y$ 轴方向上的速度分量，初值为“0”
$gy$	重力（ $Y$ 轴方向上的加速度），设为“1”

表中所列的各个变量, 小球的位置坐标“x”、“y”会随着小球的移动而改变, 小球Y轴方向上的速度分量“vy”会受到重力的影响而改变, 小球X轴方向上的速度分量“vx”及重力值“gy”则维持不变

**程序代码: MyPaint()**

```
1  //****自定义绘图函数*****
2  // 1.窗口贴图
3  // 2.计算小球速度、坐标以及判断是否碰到窗口下缘
4  void MyPaint(HDC hdc)
5  {
6      SelectObject(bufdc,bg);
7      BitBlt(mdc,0,0,640,480,bufdc,0,0,SRCCOPY);
8
9      SelectObject(bufdc,ball);
10     BitBlt(mdc,x,y,26,26,bufdc,26,0,SRCAAND);
11     BitBlt(mdc,x,y,26,26,bufdc,0,0,SRCPAINT);
12
13     BitBlt(hdc,0,0,640,480,mdc,0,0,SRCCOPY);
14
15     x += vx;           //计算 X 轴方向贴图坐标
16
17     vy = vy + gy;      //计算 Y 轴方向速度分量
18     y += vy;           //计算 Y 轴方向贴图坐标
19     if(y >= rect.bottom-26)
20     {
21         y = rect.bottom - 26;
22         vy = -vy;
23     }
24
25     tPre = GetTickCount(); //记录此次绘图时间
26 }
```

**程序说明**

MyPaint()函数先按照小球的位置坐标进行窗口贴图,然后再按照小球的移动速度和所受重力值计算下一次的贴图坐标。

(1) 第6~13行:按照坐标变量“x”与“y”的值,进行小球的透明贴图。

(2) 第15行:小球在X轴方向上的速度不变,因此只需进行累加操作来求出下次小球在X轴方向上的贴图坐标。

(3) 第17~23行:在考虑重力“gy”的影响下,先求出小球在Y轴方向上的速度,接着再以此速度求出下次小球在Y轴方向上的贴图坐标。第19~23行程序代码则判断小球落下是否碰到窗口下缘,若碰到窗口下缘则重设Y轴方向上的贴图坐标变量“y”的值,并将移动速度“vy”设为反向,产生反弹效果。

**运行结果**

程序运行结果如图6-6所示。





图 6-6

像上面的这个范例，小球从高处受到重力影响下落，与地面碰撞后弹跳至原先的高度，这是在理想的状况下按照物理中能量守恒的结果。但是在真实的状态中，物体下落会受到种种外力的影响，例如空气阻力，摩擦力等，使得物体在下坠或者弹跳的运动过程中渐渐损失自身的能量，最后变成静止状态。下一小节将继续讨论自然界中最具代表性的另一种影响物体运动的负向加速度——摩擦力。

## 6.1.4 摩擦力

摩擦力是一种作用于运动物体上的负向力。摩擦力作用于运动物体的结果，会产生与物体运动方向相反的加速度，使得物体的运动越来越慢直到静止不动为止。上一小节的范例中并没有考虑小球下坠与弹跳时摩擦力的影响，如果要让小球的运动状态符合真实世界的情况，那么就必须要加入小球运动受到摩擦力影响的效果。

这一小节的范例中将考虑在小球与地面接触时摩擦力的影响，加入使小球运动速度减慢的负向加速度，并忽略小球与空气摩擦所产生的空气阻力。图 6-7 是小球落地接触地面时，其运动方向及作用于其上的摩擦力的示意图。

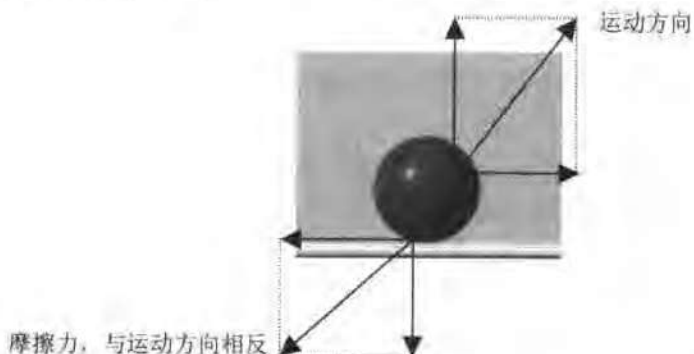


图 6-7

从图中可以看出，当小球弹起向右上方移动时，摩擦力的作用方向是左下方。因摩擦力会产生水平与垂直方向上的负向加速度，使得小球在弹跳的过程中， $X$  轴和  $Y$  轴方向上的移动速度渐渐减慢，直到最后小球静止不动为止。下面看看如何来设计这样一个比较符合真实状况的小球下落与弹跳效果。

## 》》 范例 ch6\_3

设计小球由高处下落及触地反弹的效果，并考虑小球落地时所受到的摩擦力，使得小球在几次的落地弹跳后运动速度减慢，最后呈现静止状态。

### 程序代码：全局变量声明

```

1  //全局变量声明
2  HINSTANCE    hInst;
3  HBITMAP      bg,ball;
4  HDC          hdc,mdc,bufdc;
5  HWND         hWnd;
6  DWORD        tPre,tNow,tCheck;
7  RECT         rect;
8  int          x=0,y=100,vx=8,vy=0;
9  int          gy=1,fx=-1,fy=-4;

```

### 程序说明

第8~9行：声明小球运动时所用到的相关变量，如表6-2所示。

表 6-2

变 量 名 称	说 明
x	小球X轴方向上的坐标
y	小球Y轴方向上的坐标
vx	X轴方向上的速度分量，初值为“8”
vy	Y轴方向上的速度分量，初值为“0”
gy	重力（Y轴方向上的加速度），设为“1”
fx	摩擦力在X轴方向上所产生的负向加速度
fy	摩擦力在Y轴方向上所产生的负向加速度

表中变量“fx”与“fy”是小球与地面接触时因摩擦力而产生的负向加速度，范例中便是利用这两个变量的值来改变小球落地弹跳后的运动速度的。

### 程序代码：MyPaint()

```

1  //****自定义绘图函数*****
2  // 1.窗口贴图
3  // 2.根据小球的运动状态计算速度与贴图坐标
4  void MyPaint(HDC hdc)
5  {
6      SelectObject(bufdc,bg);
7      BitBlt(mdc,0,0,750,400,bufdc,0,0,SRCCOPY);
8
9      SelectObject(bufdc,ball);
10     BitBlt(mdc,x,y,26,26,bufdc,26,0,SRCCAND);
11     BitBlt(mdc,x,y,26,26,bufdc,0,0,SRCPAINT);
12
13     BitBlt(hdc,0,0,750,400,mdc,0,0,SRCCOPY);
14
15     x += vx;          //计算X轴方向贴图坐标

```

```

16
17  vy = vy + gy;           //计算 Y 轴方向速度分量
18  y += vy;               //计算 Y 轴方向贴图坐标
19
20  if(y >= rect.bottom-26)    //小球落地
21  {
22      y = rect.bottom - 26;
23
24      //改变 X 轴方向速度分量
25      vx += fx;
26      if(vx < 0)
27          vx = 0;
28
29      //改变 Y 轴方向速度分量
30      vy += fy;
31      if(vy < 0)
32          vy = 0;
33
34      vy = -vy;
35  }
36
37  tPre = GetTickCount();    //记录此次绘图时间
38  }

```

## 程序说明

在 MyPaint() 函数中, 小球自高处落下受到重力影响和下一时刻贴图坐标计算的部分与上一小节范例中的讨论相同, 而在小球落下碰到窗口下缘部分时则会受到摩擦力所产生的负向加速度“fx”与“fy”的影响, 改变其 X 轴与 Y 轴上的运动速度分量。

(1) 第 25~27 行: 小球每次落地时, 其 X 轴方向上的运动速度“vx”受到负向加速度“fx”的影响, 速度递减。当“vx”的值递减到小于“0”时, 将其设为“0”, 这样小球在 X 轴方向上将不再移动。

(2) 第 30~32 行: 小球每次落地时, 其 Y 轴方向上的运动速度“vy”受到负向加速度“fy”的影响, 速度递减。当“vy”的值递减到小于“0”时, 将其设为“0”, 这样小球在 Y 轴方向上将不再移动。

(3) 第 34 行: 将小球 Y 轴方向上的移动速度设为负向, 产生碰到地面的反弹效果。

## 运行结果

程序运行结果如图 6-8 所示。

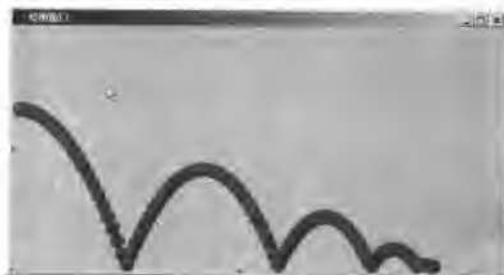


图 6-8

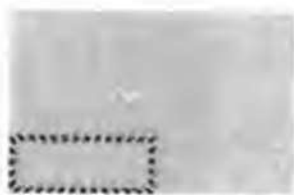
这一节中介绍了如何以程序来设计自然界中物体移动的效果。只要善加运用以上的基本物理概念并发挥一点巧思,便可将真实世界中物体的运动完整呈现在游戏中。

## 6.2 物体间的碰撞

碰撞检测在游戏中应用得较多,例如人物走到了窗口的尽头或者碰触到了其他的物体。事实上,碰撞检测的方法不只一种。有的碰撞检测是以范围来检测碰撞;有的碰撞检测则是以颜色来检测碰撞;有的碰撞检测则是以行进路线是否交叉来检测碰撞等。

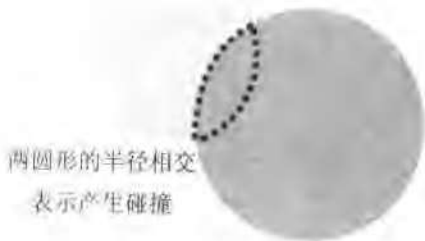
### 6.2.1 以范围检测碰撞

以范围检测碰撞的方法,适合用在规则形状可取得其范围大小的几何图形,如图 6-9 和图 6-10 所示。



两矩形的长和宽相交表示产生碰撞

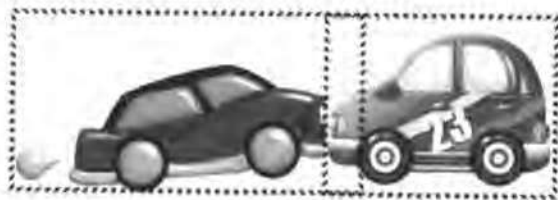
图 6-9



两圆形的半径相交表示产生碰撞

图 6-10

以范围检测碰撞的方法最简单且快速,但在制作游戏程序时,常会用到非规则形状的图形,若在允许的范围内,还是希望能够以上面的方法来检测物体是否碰撞,这样将节省许多计算的时间。例如,当两台不规则形状的车子在同样的高度上移动时,要检测两车是否碰撞的方法,只需要判断这两张车子的图片长度是否已相交,如图 6-11 所示。



两车图片的长已相交,表示两车产生碰撞

图 6-11

不过,如果两车在不同高度上移动的时候,那么就必须利用图片矩形的宽与高来检测其是否碰撞,但是这种检测碰撞的方法会产生少许的误差,如图 6-12 所示。

在图 6-12 中,在两车真正碰撞之前,已检测到两台车的矩形方框产生碰撞。如果这样的误差在允许的范围内,当然还是很乐意使用这种碰撞检测的判断方式,因为这种方式的运算速度比较快,而且程序代码比较简单。但想要能够更精确地判断非规则形状的物体是否产生碰撞,最常使用的方法还是利用颜色来判断。这部分内容将留到下一小节再来讨论,下面先来看一个利用矩形范围来检测两台车是否碰撞的范例。



图 6-12

## 》 范例 ch6\_4

这个范例中，使用了 3 张图片，分别为两张车子图案和发生碰撞时显示的提示图片（含屏蔽）。将车子的移动设定在等高的位置上，并利用车子的两个图片矩形，以两者在长度上是否产生交集来判断是否发生碰撞，而 OnTimer() 函数则主要是用来产生车子移动并判断是否发生碰撞的程序代码。

### 程序代码

```

1  int vx1=5,vx2=-5,x1=-70,x2=520,y1=165,y2=150;
2  void canvasFrame::OnTimer(UINT nIDEvent)
3  {
4      CFrameWnd::OnTimer(nIDEvent);
5      CClientDC dc(this);
6      mdc->SelectObject(car1);          //选择第一部车
7      dc.BitBlt(x1,y1,196,66,mdc,0,0,SRCCOPY);
8      //贴上第一部车
9      mdc->SelectObject(car2);          //选择第二部车
10     dc.BitBlt(x2,y2,140,80,mdc,0,0,SRCCOPY);
11     //贴上第二部车
12     if(x1+196>x2)                    //判断矩形是否相交
13     {
14         mdc->SelectObject(bomb);
15         dc.BitBlt(x2-100,y2,187,100,mdc,0,100,SRCPAINT);
16         dc.BitBlt(x2-100,y2,187,100,mdc,0,0,SRCCAND);
17         KillTimer(nIDEvent);
18     }
19     x1+=vx1;                          //第一部车下次贴图位置
20     x2+=vx2;                          //第二部车下次贴图位置
21 }
    
```

### 程序说明

(1) 第 1 行：全局变量声明，各个声明变量的意义如表 6-3 所示。

表 6-3

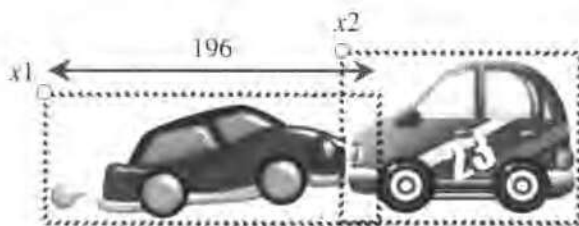
变量名称	说 明
vx1	第一部车的移动速度，设为“5”，为由左向右移动
vx2	第二部车的移动速度，设为“-5”，为由右向左移动
x1	第一部车 X 轴方向的位置，一开始设为“-70”，在最左边

(续表)

变量名称	说明
x2	第二部车 X 轴方向的位置, 一开始设为“520”, 在最右边
y1	第一部车 Y 轴方向的位置, 固定设为“165”
y2	第二部车 Y 轴方向的位置, 固定设为“150”

(2) 第 6~11 行: 按顺序利用  $(x1, y1)$ 、 $(x2, y2)$  坐标, 在窗口中贴上第一、二部车, 产生车子移动的效果。

(3) 第 12 行: 判断两矩形在长度上是否产生交集。如果是, 则表示发生碰撞, 运行显示提示图案并停止定时器的程序代码, 如图 6-13 所示。



若“ $x1+196$ ”大于“ $x2$ ”, 则表示已发生碰撞

图 6-13

(4) 第 19、20 行: 按照“ $vx1$ ”与“ $vx2$ ”的值来计算下一次两车显示的坐标“ $x1$ ”与“ $x2$ ”。

#### 运行结果

程序运行结果如图 6-14 和图 6-15 所示。

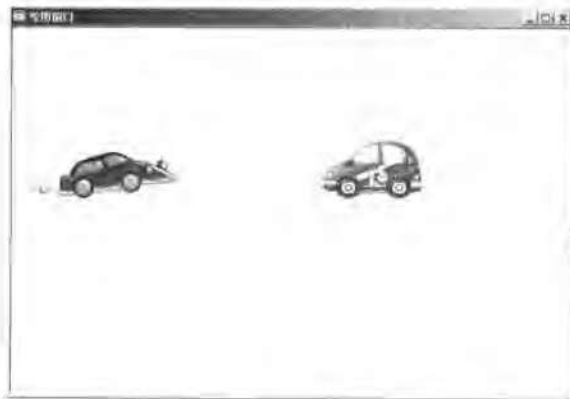


图 6-14



图 6-15

#### 技巧

在这个范例中, 因为使用的位图底色不同, 因此制作透明图时使用了另一种运算方法。来源位图的内容如图 6-16 所示。

因为位图的底色与屏蔽图的颜色黑色与白色部分与前面介绍的刚好相反, 因此制作透明的流程图, 便有了不同, 其步骤为:

- (1) 将屏蔽与背景图做“OR”运算。
- (2) 将要贴上的图与上一个结果做“AND”运算。

## 撞到了



图 6-16

其中产生透明图的运算方法在此就不多说了。如要详细了解这方面的内容，可参照第 2 章中的内容说明自行演练。

### 6.2.2 以颜色检测碰撞

以颜色检测碰撞的方法比较麻烦，但这种碰撞检测方法可以很精确地判断两个非规则形状物体是否真的发生碰撞，假设目前会发生碰撞的情况如图 6-17 所示。



图 6-17

检测车子是否进入树林中的方法便是检测车子是否与树林发生碰撞，那么应该如何利用颜色来判断车子是否与树林发生碰撞呢？由图中是看不出任何蛛丝马迹的，因为它无法以任何颜色为基准点来计算出是否发生碰撞，现在换了一张图来试试看，如图 6-18 所示。



图 6-18

图中黑色部分与前一个树林是一模一样的。之所以要将树林改成黑色，是为了要把它当做是一张“暗图”，以便利用颜色来判断是否发生碰撞。

当车子与树林发生碰撞时，车头的部分被黑色所遮蔽。这里要判断是否碰撞的要点就在于黑色部分与车子是否会产生交集，但是又如何判断车子与黑色部分有交集呢？因为黑色与任何颜色做“AND”运算其结果还是黑色，所以如果我们以车子的颜色来跟目前所在位置上的“暗图”做 AND 运算之后，即会有黑色的结果产生，便表示车子与树林发生了碰撞（前提是车子图案中不可以有纯黑色）。

接下来就以颜色判断的方法来检测是否产生碰撞，其概念如图 6-19 和图 6-20 所示。



图 6-19



图 6-20

下面来看一个利用颜色检测碰撞的范例。在范例中, 车子由右至左移动, 当进入树林时 (即与树林产生碰撞), 会出现提示信息。

## » 范例 ch6\_5

### 程序代码

```

1  int vx=-5,x=700,i,b,g,r;
2  canvasFrame::canvasFrame()
3  {
4  /* 建立窗口的程序代码 (略) */
5  //以下程序代码建立各个所需要的 DC
6  mdc = new CDC;
7  mdc1 = new CDC;
8  mdc2 = new CDC;
9  mdc->CreateCompatibleDC(&dc);
10 mdc1->CreateCompatibleDC(&dc);
11 mdc2->CreateCompatibleDC(&dc);
12 //以下程序代码建立与加载所要用的位图对象
13 car = new CBitmap;
14 forest = new CBitmap;
15 mask = new CBitmap;
16 temp = new CBitmap;
17 dark = new CBitmap;
18 car->m_hObject = (HBITMAP)::LoadImage(NULL,"car.bmp",
19 IMAGE_BITMAP,112,64,LR_LOADFROMFILE);
20 forest->m_hObject = (HBITMAP)::LoadImage(NULL,"forest.bmp",
21 IMAGE_BITMAP,800,300,LR_LOADFROMFILE);
22 mask->m_hObject = (HBITMAP)::LoadImage(NULL,"mask.bmp",
23 IMAGE_BITMAP,800,300,LR_LOADFROMFILE);
24 temp->CreateCompatibleBitmap(&dc,rect.right,rect.bottom);

```



```

25 dark->CreateCompatibleBitmap(&dc,112,64);
26 mdc->SelectObject(temp);    //设定 mdc 中存储位图的格式
27 car->GetObject(sizeof(BITMAP),&bm);
28 px = new unsigned char[bm.bmHeight*bm.bmWidthBytes];
29 car->GetBitmapBits(bm.bmHeight*bm.bmWidthBytes,px);
30 mdc2->SelectObject(dark);    //设定 mdc2 中存储位图格式
31 )

```

## 程序说明

- (1) 第 1 行：全局变量声明。“vx”与“x”分别为车子在 X 轴方向上的速度以及初始化位置；“b”、“g”、“r”用来记录颜色值；“i”为计数用变量。
- (2) 第 6~8 行：此程序中使用多个暂存 DC 与位图对象。表 6-4 是 3 个暂存 DC 的用途说明。

表 6-4

DC 名称	说 明
mdc	在此 DC 上制作透明
mdc1	此 DC 用来取得位图对象
mdc2	存放车子所在位置区域中暗图的影像

各个位图对象所代表的意义则如表 6-5 所示。

表 6-5

位 图	说 明
car	车子图
forest	树林图
mask	树林的屏蔽图
temp	mdc 中的位图
dark	mdc2 中的位图，存放车子所在区域中，暗图的影像

- (3) 第 24、25 行：建立位图“temp”与“dark”的格式。
- (4) 第 26、30 行：设定“mdc”与“mdc2”中要存储的位图格式分别为“temp”与“dark”。
- (5) 第 27~29 行：取得车子图中的所有像素点的颜色并存入数组“px”中，利用“px”数组中的值在程序中通过颜色来判断是否发生碰撞。

程序中设定了一个定时器，OnTimer()函数中的程序代码主要分为两个部分：制作透明并处理车子的移动；进行颜色值的计算来判断是否发生碰撞。

程序代码如下。

## 程序代码

```

1 void canvasFrame::OnTimer(UINT nIDEvent)
2 {
3     CFrameWnd::OnTimer(nIDEvent);
4     CClientDC dc(this);
5     mdc->BitBlt(0,0,800,300,mdc1,0,0,WHITENESS);
6     mdc1->SelectObject(car);
7     mdc->BitBlt(x,160,112,64,mdc1,0,0,SRCCOPY);
8     mdc1->SelectObject(mask);

```

```

9   mdc2->BitBlt(0,0,112,64,mdc1,x,160,SRCCOPY);
10  dark->GetObject(sizeof(BITMAP),&bm);
11  unsigned char *px1 = new unsigned char[bm.bmHeight*bm.bmWidthBytes];
12  dark->GetBitmapBits(bm.bmHeight*bm.bmWidthBytes,px1);
13  if (bm.bmBitsPixel != 32)
14  {
15      return;
16  }
17  int rgb_b,PixelBytes,tx,ty;
18  PixelBytes = bm.bmBitsPixel / 8;
19  for (ty=0;ty<bm.bmHeight;ty++)
20  {
21      for (tx=0;tx<bm.bmWidth;tx++)
22      {
23          rgb_b = ty*bm.bmWidthBytes+tx*PixelBytes;
24          if(px[rgb_b] != 255 && px[rgb_b+1] != 255 && px[rgb_b+2] !=255)
25          {
26              b = px[rgb_b] & px1[rgb_b];
27              g = px[rgb_b+1] & px1[rgb_b+1];
28              r = px[rgb_b+2] & px1[rgb_b+2];
29              if(b == 0 && g==0 && r==0)
30              {
31                  mdc->TextOut(300,50,"车子在树林中...");
32                  break;
33              }
34          }
35      }
36  }
37  delete px1;
38  mdc->BitBlt(0,0,800,300,mdc1,0,0,SRCCAND);
39  mdc1->SelectObject(forest);
40  mdc->BitBlt(0,0,800,300,mdc1,0,0,SRCPAINT);
41  dc.BitBlt(0,0,800,300,mdc,0,0,SRCCOPY);
42  if(x<-112) //判断车子是否已完全没入窗口中
43  KillTimer(nIDEvent); //停止定时器
44  x+=vx; //计算下一次的贴图坐标
45  }

```

#### 程序说明

- (1) 第5行：清除上一次的贴图。前一次所选择的位图对象为“forest”，其大小为窗口大小，因此将其转换为白色（WHITENESS）可清除上一次的贴图内容。
- (2) 第6、7行：选择“car”位图，并贴入“mdc”中当做背景图。
- (3) 第8、38行：选择树林的屏蔽图“mask”，并与“mdc”的内容做“SRCCAND”运算贴图。
- (4) 第39、40行：选择树林位图“forest”，并与“mdc”的内容做“SRCPAINT”运算贴图，这样便在“mdc”中产生了所要的透明图。
- (5) 第41行：将“mdc”的内容复制到窗口中。

(6) 第 17~36 行：利用颜色值检测是否碰撞的程序代码。若检测到即表明发生碰撞，则新增提示信息到 `mdc` 中并显示在窗口上。

(7) 第 24 行：检测 `px[rgb_b]`、`px[rgb_b+1]`、`px[rgb_b+2]`，即 `rgb_b` 像素点的色彩值是否为白色（3 个元素都为 255）。如果是白色的话，则该点是车子图的白色部分，便跳过不做任何处置；若不为白色，则该点是车子中的点，接下来便与 `px1` 数组中对应的元素做“AND”运算。如果 `b`、`g`、`r` 的值都为“0”，就表示运算结果为“黑色”，也就表示车子与树林发生了碰撞。

## 运行结果

程序运行结果如图 6-21 和图 6-22 所示。



图 6-21



图 6-22

在这个程序中，每次当车子移动时必须重新做透明以获得位图所有像素的色彩值，并加以运算然后再判断是否产生碰撞。

## 6.2.3 以行进路线检测碰撞

以行进路线来检测是否碰撞的方法主要用于检测两个移动的物体或者移动物体与平面是否发生碰撞，如图 6-23 和图 6-24 所示。

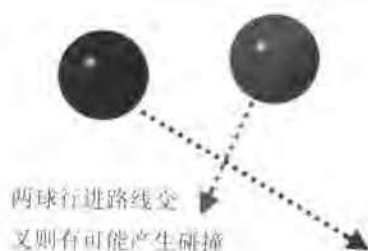


图 6-23

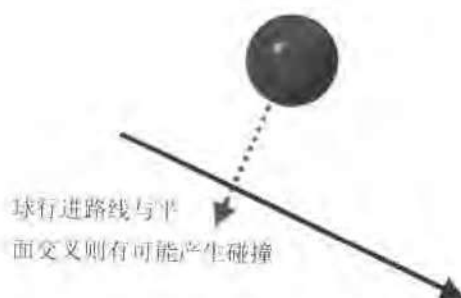


图 6-24

如图 6-23 和图 6-24 所示, 不管两颗球的行进方向、或者是平面, 它们各自都加上了一个箭头, 表示其向量。

下面以向量来判断一个具有速度值的小球是否与斜面(非水平与垂直)会发生碰撞, 这里先假设小球目前的时刻与下一个时刻的圆心位置分别为  $P_3$  与  $P_4$ , 而斜面的起点与终点为  $P_1$  与  $P_2$ , 原点为  $O$ , 若小球与平面发生碰撞则碰撞点为  $C$ , 其碰撞过程如图 6-25 所示。

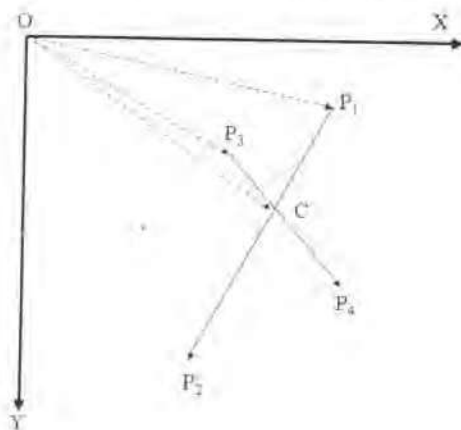


图 6-25

从图中我们可推导出如下的式子:

$$OP_1C \text{ 中: } OC = OP_1 + P_1C = OP_1 + mP_1P_2$$

$$OP_3C \text{ 中: } OC = OP_3 + P_3C = OP_3 + nP_3P_4$$

$$\Rightarrow OP_1 + mP_1P_2 = OP_3 + nP_3P_4$$

若交点  $C$  在两向量之中, 则在上面的式子中,  $m$  与  $n$  的值会介于  $0 \sim 1$  之间, 其值代表球与斜面是否发生碰撞。

假设斜面的起点坐标  $P_1$  为  $(a, b)$ , 而向量  $P_2 - P_1$  可得  $(Lx, Ly)$ , 小球圆心目前的坐标为  $(c, d)$ , 其速度向量为  $(Vx, Vy)$ , 代入上式得:

$$(a, b) + m(Lx, Ly) = (c, d) + n(Vx, Vy)$$

$$\rightarrow X \text{ 轴方向的向量: } a + mLx = c + nVx$$

$$Y \text{ 轴方向的向量: } b + mLy = d + nVy$$

$$\rightarrow m = [Vx(b-d) + Vy(c-a)] / (LxVy - LyVx)$$

$$n=[Lx(d-b)+Ly(a-c)]/(VxLy-VyLx)$$

在导出了  $m$  与  $n$  的结果之后, 在程序中若要判断小球与斜面是否会发生碰撞, 只需将其移动的路径和斜面的向量与起点坐标代入上面的方程式中, 然后判断  $m$  与  $n$  是否都介于  $0 \sim 1$  之间, 就可得知是否发生碰撞。

## 6.2.4 与斜面碰撞后的速度

由上一小节的说明, 已经可以判断出小球是否与斜面发生碰撞。如果小球与斜面发生碰撞的话, 那么如何求得碰撞之后的速度则成为这一小节所要讨论的重点。

在这里, 利用向量的方式来求得碰撞后的速度是最方便的, 但其计算的过程又比前一节的内容复杂一些。图 6-26 是小球与斜面发生完全碰撞并反弹的示意图。

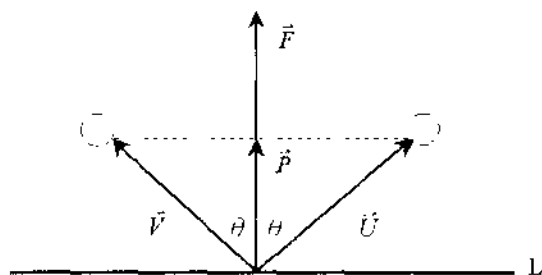


图 6-26

图中,  $\vec{F}$  是斜面  $L$  的一个法向量,  $\vec{V}$  与  $\vec{U}$  则分别是小球与斜面碰撞前与碰撞后的速度, 入射角与反射角为  $\theta$ ,  $\vec{P}$  为  $\vec{V}$  在  $\vec{F}$  上负向的投影, 由上图可得知  $\vec{U} = \vec{V} + 2\vec{P}$ , 其意义如图 6-27 所示。

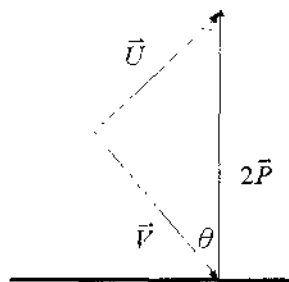


图 6-27

为了求得碰撞后的速度  $\vec{U}$ , 现在的重点是如何算出  $\vec{P}$ , 因为  $\vec{P}$  为  $\vec{V}$  在  $\vec{F}$  上负向的“投影”, 因此  $\vec{P}$  便是  $\vec{V}$  在法向量  $\vec{F}$  上的“负投影长”乘以“单位法向量”, 下面来看看求得碰撞后速度  $\vec{U}$  的计算过程:

$$\vec{V} \text{ 在 } \vec{F} \text{ 上的负投影长: } |\vec{P}| = |-\vec{V} \cdot \cos \theta|$$

$$\vec{F} \text{ 的单位法向量: } \vec{f} = \frac{\vec{F}}{|\vec{F}|}$$

$$\Rightarrow \vec{P} = \vec{P} \cdot \vec{f} = |-\vec{V}| \cdot \cos \theta \cdot \frac{\vec{F}}{|\vec{F}|}$$

接着需要算出  $\cos\theta$  的值, 其内积公式为:

$$\begin{aligned}\vec{a} \cdot \vec{b} &= |\vec{a}| |\vec{b}| \cos\theta \\ \Rightarrow -\vec{V} \cdot \vec{F} &= |-\vec{V}| |\vec{F}| \cos\theta \\ \Rightarrow \cos\theta &= \frac{-\vec{V} \cdot \vec{F}}{|-\vec{V}| |\vec{F}|}\end{aligned}$$

将  $\cos\theta$  代入上式, 得:

$$\begin{aligned}\vec{P} &= |-\vec{V}| \times \frac{-\vec{V} \cdot \vec{F}}{|-\vec{V}| |\vec{F}|} \times \frac{\vec{F}}{|\vec{F}|} = \frac{-\vec{V} \cdot \vec{F}}{|\vec{F}|^2} \times \vec{F} \\ \text{又 } \vec{U} &= \vec{V} + 2\vec{P} \\ \vec{U} &= \vec{V} - 2 \times \frac{-\vec{V} \cdot \vec{F}}{|\vec{F}|^2} \times \vec{F}\end{aligned}$$

假设小球射入的速度  $\vec{V}$  为  $(V_x, V_y)$ , 斜面法向量  $\vec{F}$  为  $(F_x, F_y)$ , 则小球反弹后的速度即为  $\vec{U}$ 。将  $\vec{V}$  与  $\vec{F}$  的值代入上式, 便可求得小球反弹后在  $X$  轴和  $Y$  轴上的速度分量  $(U_x, U_y)$ 。下面直接以程序代码来表示  $U_x$  和  $U_y$  的计算结果:

$$\begin{aligned}U_x &= V_x - 2 * (F_x * V_x + F_y * V_y) * F_x / (\text{pow}(F_x, 2) + \text{pow}(F_y, 2)); \\ U_y &= V_y - 2 * (F_x * V_x + F_y * V_y) * F_y / (\text{pow}(F_x, 2) + \text{pow}(F_y, 2));\end{aligned}$$

上面的式子中用到了计算数值次方的函数“pow”, 此函数定义在“math.h”中, 其中输入的参数分别为要计算的“底数”与“指数”。

接下来要介绍的这个范例中, 在窗口里加入了代表斜面的两条线段, 当小球在窗口中移动的时候, 利用前面介绍的向量概念来判断其是否与斜面或者窗口边缘发生碰撞, 并计算出碰撞后行进的速度与方向。

## » 范例 ch6\_6

在这个范例中, 同样使用一个定时器来处理小球的移动, 其中包括检测是否发生碰撞和在窗口中重绘小球图案等内容, 程序代码如下。

### 程序代码

```
1  int x=0,y=0,xlast,ylast,ox,oy;
2  float vx=21,vy=21;
3  float m,n;
4  void canvasFrame::OnTimer(UINT nIDEvent)
5  {
6  CFrameWnd::OnTimer(nIDEvent);
7  CClientDC dc(this);
8  dc.BitBlt(xlast,ylast,39,39,mdc,0,0,WHITENESS);
9  dc.BitBlt(x,y,39,39,mdc,0,0,SRCCOPY);
10 xlast = x;           //上次贴图 x 坐标
11 ylast = y;           //上次贴图 y 坐标
12 x += (int)vx;         //计算新的贴图 x 坐标
13 y += (int)vy;         //计算新的贴图 y 坐标
```

```

14  ox = x+20;                                //圆心 X 坐标
15  oy = y+20;                                //圆心 Y 坐标
16  collide(rect.right,20,400,rect.bottom);    //是否碰到左边斜面
17  collide(400,rect.bottom,0,200);           //是否碰到右边斜面
18  if(y < 0)                                  //是否碰到上缘
19  {
20    vy = -vy;
21    y = 0;
22  }
23  if(x < 0)                                  //是否碰到左缘
24  {
25    vx = -vx;
26    x = 0;
27  }
28  dc.MoveTo(rect.right,20);                  //画线
29  dc.LineTo(400,rect.bottom);
30  dc.LineTo(0,200);
31  }

```

## 程序说明

(1) 第 1 行：声明的全局变量说明见表 6-6。

表 6-6

变 量	说 明
x, y	待贴小球图的 X 与 Y 坐标，起始坐标为 (0,0)
xlast, ylast	上一次的贴图坐标，这两个变量用来清除上次贴图的内容
ox, oy	小球圆心坐标，以此坐标来判断是否发生碰撞

(2) 第 2 行：声明小球速度变量“vx”与“vy”，其初始化速度都设为“21”。

(3) 第 3 行：声明变量“m”与“n”，其目的是用来判断是否发生碰撞的系数。

(4) 第 8、9 行：清除上一次的贴图，并在新的坐标上贴上小球图。

(5) 第 10~15 行：取得各个需要的坐标，其中第 12、13 行程序代码为按照小球的速度求得下一次的贴图坐标。

(6) 第 16、17 行：调用自定义函数 collide 来判断小球是否与两斜面发生碰撞，并计算出碰撞后小球在 X、Y 轴上的速度分量，其中输入的 4 个参数分别为斜面的“起点 X 坐标”、“起点 Y 坐标”、“终点 X 坐标”、“终点 Y 坐标”，这个函数中的内容稍后再做说明。

(7) 第 18~27 行：判断小球是否与窗口上缘或左缘发生碰撞，若发生碰撞则同样重设小球的速度以及 X 与 Y 坐标。

(8) 第 28~30 行：在屏幕上绘制代表斜面的两线段。

在自定义函数 collide 中，会按照线段的起点与终点来计算斜面的向量与法向量，并由前面我们推导出的公式求得小球碰撞后的速度，程序代码如下。

## 程序代码

```

1  void canvasFrame::collide(int startX,int startY,int endX,int endY)
2  {
3  float va,vb;                                //碰撞后速度
4  float lx,ly,fx,fy;

```

```

5  lx = (float)endX-startX;           // X 轴向量
6  ly = (float)endY-startY;           // Y 轴向量
7  fx = (float)startY-endY;           // X 轴法向量
8  fy = (float)endX-startX;           // Y 轴法向量
9  m = (vx*(startY-oy)+vy*(ox-startX))/(lx*vy-ly*vx);
10 n = (lx*(oy-startY)+ly*(startX-ox))/(vx*ly-vy*lx);
11 if(m>=0 && m<=1 && n>=0 && n<=1)
12 {
13     va = (float)(-2*(fx*vx+fy*vy)*fx/(pow(fx,2)+pow(fy,2))+vx);
14     vb = (float)(-2*(fx*vx+fy*vy)*fy/(pow(fx,2)+pow(fy,2))+vy);
15     vx = va;
16     vy = vb;
17 }
18 }

```

#### 程序说明

(1) 第5~8行: 求得线段的向量与法向量, 若输入的点坐标为 (startX,startY) 与 (endX,endY), 则线段的向量与法向量分别为 (endX-startX,endY-startY) 与 (startY-endY,endX-startX)。

(2) 第9、10行: 按照前一小节的公式来求得  $m$  与  $n$  的值, 这两个数值是用来判断小球是否与斜面发生碰撞。

(3) 第11行: 判断  $m$  与  $n$  是否介于0~1之间, 若条件成立, 则表示发生碰撞。

(4) 第13、14行: 根据公式算出碰撞后小球在  $X$  轴与  $Y$  轴上的速度分量 “ $va$ ” 与 “ $vb$ ”, 并重设 “ $vx$ ” 与 “ $vy$ ” 的值。

#### 运行结果

程序运行结果如图 6-28 所示。

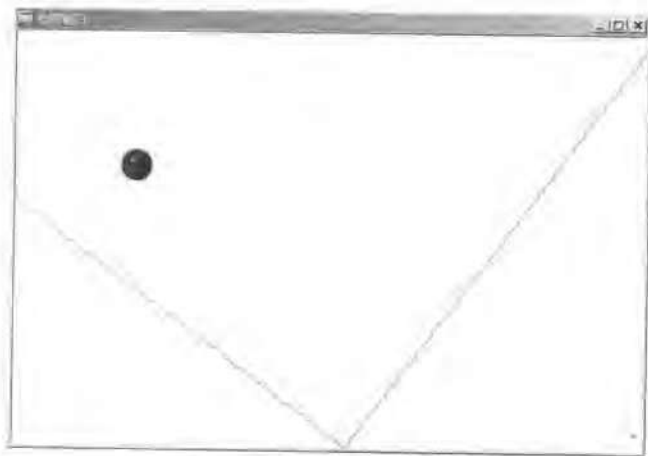


图 6-28

## 6.3 粒子的应用

可以把“粒子”想像成颗粒状的物体, 如雪花、一块炸弹碎片、一颗沙子或烟雾中的粒子等特殊效果。许多的粒子组合起来, 就变成了一些在现实生活中可以看到的物理及自然现象了。



## 6.3.1 粒子的定义

要使用粒子，首先必须了解粒子的用途与特性，然后再定义出一个适合的粒子类型。在 C 语言中定义某种粒子的类型是相当容易的，只要自定义一个粒子的结构就行，如下面的这个自定义结构“snow”便是用来代表“雪花”粒子的。

```
struct snow
{
    int x;           //雪花所在的 X 坐标
    int y;           //雪花所在的 Y 坐标
    BOOL exist;      //是否存在
};
```

目前使用“x”与“y”的成员来表示粒子所在的位置，“exist”则是用来代表该粒子是否还存在，若以雪花特效为例，当雪花粒子飘落到地上时，该粒子的“exist”成员就可能设为“false”了。

定义出粒子的结构之后，便可以在程序中建立一个粒子类型的数组，以便产生许多个粒子供程序使用，如下面的这行声明便是建立了一个大小为 50 的 snow 数组，其中的各个元素便是代表一片片的雪花。

```
snow flakes[50];
```

这就是使用自定义结构建立粒子类型的方法，事实上自定义结构还可定义其他许多类型的对象。在本书后面的内容中，将可以看到使用自定义结构来定义了子弹、怪物等多种的对象类型。

## 6.3.2 雪花纷飞

“下雪”可以说是相当常见的自然现象，而在程序中要表现下雪时雪花纷飞的情景，使用粒子来表现可以说是最恰当不过的了。

下面这个范例，便是以粒子来模拟下雪的景象。程序开始运行后会不断地落下雪花，一直到出现雪花数量的上限时（设为 50）便不再继续增加，而当雪花落到地上时，便重新设定该雪花粒子的显示位置回到上方，以产生雪花消失与新的雪花落下的效果。

### 《范例》 ch6\_7

在雪花的特效程序里，OnTimer()函数是整个范例的重点，其中当雪花的总数小于 50 时，便会自动产生雪花粒子，而且每次会取出粒子数组中所有的元素，按照粒子的所在位置将其显示在屏幕上并重新计算新的坐标。下面先来看看程序代码。

#### 程序代码

```
1  struct snow
2  {
3      int x;           //雪花所在的 X 坐标
4      int y;           //雪花所在的 Y 坐标
5      BOOL exist;      //是否存在
6  };
7  int i,count;
```

```

8   snow flakes[50];
9   void canvasFrame::OnTimer(UINT nIDEvent)
10  {
11  if(count<50)
12  {
13      flakes[count].x = rand()%rect.right;
14      flakes[count].y = 0;
15      flakes[count].exist = true;
16      count++; //累加粒子总数
17  }
18  CCClientDC dc(this);
19  mdc1->SelectObject(Lgbmp);
20  mdc->BitBlt(0,0,rect.right,rect.bottom,mdc1,0,0,SRCCOPY);
21  for(i=0;i<50;i++)
22  {
23      if(flakes[i].exist)
24      {
25          mdc1->SelectObject(mask);
26          mdc->BitBlt(flakes[i].x,flakes[i].y,20,20,mdc1,0,0,SRCCAND);
27          mdc1->SelectObject(snow);
28          mdc->BitBlt(flakes[i].x,flakes[i].y,20,20,mdc1,0,0,SRCPAINT);
29          if(rand()%2==0)
30              flakes[i].x+=3;
31          else
32              flakes[i].x-=3;
33          flakes[i].y+=10;
34          if(flakes[i].y > rect.bottom) //落到底部
35          {
36              flakes[i].x = rand()%rect.right;
37              flakes[i].y = 0;
38          }
39      }
40  }
41  dc.BitBlt(0,0,rect.right,rect.bottom,mdc,0,0,SRCCOPY);
42  CFrameWnd::OnTimer(nIDEvent);
43  }

```

#### 程序说明

- (1) 第1~6行：定义代表雪花粒子的结构。
- (2) 第8行：声明大小为50的粒子数组。
- (3) 第11~17行：当粒子总数小于50时，使产生新的粒子。其中第13行的程序代码将粒子的X坐标设为窗口中水平方向上的任意位置；第14行的程序代码设定其Y坐标为“0”，即表示在窗口的最上缘；第15行的程序代码设定此粒子是否存在；第16行的程序代码则是在产生一个粒子之后累加粒子的总数。
- (4) 第19、20行：在暂存DC中贴上背景图。
- (5) 第21~40行：取出“flakes”数组中的各个粒子元素，并按照其各自的位置将雪花图片贴到DC中，然后再计算其下一轮的显示位置。

(6) 第 23~28 行: 判断当粒子存在时, 按其位置坐标 (flakes[i].x, flakes[i].y) 来进行贴图。

(7) 第 29~31 行: 以随机数计数下一次的 X 坐标, 若 “rand()%2==0”, 则下一次粒子的显示坐标为 “flakes[i].x+=3” 向右移动 3 个像素点, 否则设定其向左移动 3 个像素点, 这样才会有雪花左右飘动的效果。

(8) 第 33 行: 设定粒子 Y 轴方向的显示位置为 “flakes[i].y+=10”, 每次向下移动 10 个像素点。

(9) 第 36、37 行: 当雪花落到窗口底部的时候 (flakes[i].y>rect.bottom), 程序代码重设其下一次的显示坐标为在窗口上缘且水平方向为任意点的位置上。

(10) 第 41 行: 在屏幕上显示最后的画面。随着每次雪花位置的改变以及利用定时器不断地更新屏幕的显示, 便可以看到雪花纷飞的效果了。

## 运行结果

程序运行结果如图 6-29 所示。



图 6-29

## 6.3.3 放烟火

您一定看过夜空中五彩缤纷的烟火吧, 其实放烟火的原理跟物体爆炸的原理是一样的, 可将每一颗烟火爆炸后所出现闪亮的碎片视为一颗粒子, 因此同样可以运用粒子的方法在程序中模拟烟火的爆炸画面。

这一小节将给出一个仿真烟火爆炸的范例, 其中烟火的爆炸点是在窗口中由随机数所产生的位置。在发生爆炸后会出现许多黄色的爆炸粒子以不同的速度向四方飞散而去, 当粒子飞出窗口外或者超过一定的存在时间后便会消失。当每一次爆炸所出现的粒子全部消失之后, 便重新出现爆炸的画面, 以产生不断施放烟火的效果。

## 》》范例 ch6\_8

下面先来看看爆炸碎片“火球”粒子结构的定义与全局变量声明的程序代码。

### 程序代码

```

1  struct fireball
2  {

```

```

3  int x;                //火球所在的 X 坐标
4  int y;                //火球所在的 Y 坐标
5  int vx;               //火球 X 方向的速度
6  int vy;               //火球 Y 方向的速度
7  int lasted;           //火球的存在时间
8  BOOL exist;           //是否存在
9  };
10 int i,count;
11 int x,y;
12 fireball fireball[50];

```

#### 程序说明

(1) 第1~9行: 定义火球粒子的结构, 其中由于粒子飞散时也有速度, 因此定义“vx”和“vy”来代表粒子的速度; “lasted”成员则用来代表粒子存在了多久。在这个范例中是以随机数来决定粒子的速度, 因此有些粒子移动速度可能会很慢, 利用“lasted”来记录粒子存在的时间, 这样可以在粒子存在超过一定时间后将其删除。

(2) 第10行: 声明“count”变量, 其目的是用来记录目前火球粒子的总数。此数值会随着火球的消失而递减, 当火球全部消失时则“count”值为“0”, 此时重新出现爆炸点以及新的火球粒子, “count”值重设为“50”。

(3) 第11行: 声明“x”和“y”变量, 用来代表爆炸点所在的位置, 所有火球一开始的位置即是在爆炸点上。

(4) 第12行: 声明大小为“50”的火球粒子数组。

在 OnTimer() 函数中, 当火球的总数等于“0”时, 便重新产生一个爆炸点与各火球粒子, 此后程序计算每一个火球粒子的位置坐标并将其显示在屏幕上。

#### 程序代码

```

1  void canvasFrame::OnTimer(UINT nIDEvent)
2  {
3  if(count == 0)                //新增爆炸点
4  {
5      x=rand()%rect.right;
6      y=rand()%rect.bottom;
7      for(i=0;i<50;i++)        //产生火球粒子
8      {
9          fireball[i].x = x;
10         fireball[i].y = y;
11         fireball[i].lasted = 0;
12         if(i%2==0)
13         {
14             fireball[i].vx = -rand()%30;
15             fireball[i].vy = -rand()%30;
16         }
17         if(i%2==1)
18         {
19             fireball[i].vx = rand()%30;
20             fireball[i].vy = rand()%30;
21         }

```

```

22         if(i%4==2)
23         {
24             fireball[i].vx = -rand()%30;
25             fireball[i].vy = rand()%30;
26         }
27         if(i%4==3)
28         {
29             fireball[i].vx = rand()%30;
30             fireball[i].vy = -rand()%30;
31         }
32         fireball[i].exist = true;
33     }
34     count = 50;
35 }
36 CClientDC dc(this);
37 mdc1->SelectObject(bgbmp);
38 mdc->BitBlt(0,0,rect.right,rect.bottom,mdc1,0,0,SRCCOPY);
39 for(i=0;i<50;i++)
40 {
41     if(fireball[i].exist)
42     {
43         mdc1->SelectObject(mask);
44         mdc->BitBlt(fireball[i].x,fireball[i].y,10,10,mdc1,0,0,SRCCAND);
45         mdc1->SelectObject(fire);
46         mdc->BitBlt(fireball[i].x,fireball[i].y,10,10,mdc1,0,0,SRCPAINT);
47         fireball[i].x+=fireball[i].vx;
48         fireball[i].y+=fireball[i].vy;
49         fireball[i].lasted++;
50         if(fireball[i].x<=-10 || fireball[i].x>rect.right || fireball[i].y<=-10
51            || fireball[i].y>rect.bottom || fireball[i].lasted>50)
52         {
53             fireball[i].exist = false;    //删除火球粒子
54             count--;                      //递减火球总数
55         }
56     }
57 }
58 dc.BitBlt(0,0,rect.right,rect.bottom,mdc,0,0,SRCCOPY);
59 CFrameWnd::OnTimer(nIDEvent);
60 }

```

## 程序说明

(1) 第3~35行：新增爆炸点与火球粒子的程序代码，其中第5、6行的程序代码是以随机数来取得一个爆炸点的坐标(x,y)。

(2) 第7~33行：产生50个火球粒子并初始化各个粒子的特性。

(3) 第9、10行：设定粒子的起始位置即爆炸点所在的位置；第11行的程序代码设定该粒子存在的时间为“0”。

(4) 第12~31行：设定各个粒子的速度，按粒子的编号“i”来决定粒子速度的正负方向，以朝4个象限位置移动。设定粒子在X轴与Y轴方向上的速度时，以rand()函数产生随机数并除以30，

所得的移动速度便介于 0~30 之间。

(5) 第 32 行: 将粒子的“exist”成员设为“true”, 表示粒子存在。

(6) 第 34 行: 在产生所有粒子之后, 将“count”设为“50”, 代表目前有 50 颗火球粒子。

(7) 第 37、38 行: 先在暂存 DC 中贴上背景的星空图案。

(8) 第 39~57 行: 在“for”循环中, 第 41 行的程序代码判断粒子是否还存在, 若粒子还存在, 则第 43~46 行的程序代码根据其坐标 (fireball[i].x, fireball[i].y) 贴上火球图案; 第 47、48 行的程序代码按照粒子的速度计算下一次的贴图坐标; 第 49 行的程序代码在每进行一次贴图之后, 将粒子的存在时间累加“1”。

(9) 第 50、51 行: 判断式会判断粒子是否已跑出窗口外或者其存在时间是否大于 50, 若任意一个条件成立, 则第 53 行的程序代码将该粒子设为不存在, 而第 54 行的程序代码则递减火球粒子的总数。

这样不断地运行 OnTimer() 函数, 将会看到不断产生爆炸粒子飞散的景象。

#### 运行结果

程序运行结果如图 6-30 和图 6-31 所示。



以爆炸点为中心向四周散开的火球粒子

图 6-30



每一个粒子以各自的速度向四周飞散

图 6-31

这一章里介绍了利用程序来设计物理现象的技巧，如果仔细地观察生活周围的事物并发挥巧思，相信必定能够在程序中设计出不凡的功能与效果。

## 课后重点整理

- 速度是物体在各个方向上“速度分量”的合成。
- 匀速运动是指物体在每一个时刻的速度都是相同的，即“ $V_x$ ”与“ $V_y$ ”都保持不变。
- 只要物体移动，其运动的速度或方向都会随着时间而改变，这个物体的运动便属于加速度运动。
- 地心引力也就是重力，是一个在垂直方向（Y轴方向）、值大约为  $9.8\text{m/s}$ 、方向向下的加速度。
- 摩擦力是一种作用于运动物体上的负向力。摩擦力作用于运动物体上的结果，会产生与物体运动方向相反的加速度，使得物体的运动越来越慢直到静止不动为止。
- 碰撞检测的方法不止一种。有的碰撞检测以范围来检测碰撞；有的碰撞检测以颜色来检测碰撞；有的碰撞检测则以行进路线是否交叉来检测碰撞等。
- 以范围检测碰撞的方法，适用于规则形状、可取得其范围大小的几何图形。
- 以颜色检测碰撞方法可以很精确地判断两个非规则状物体是否真的发生碰撞。
- 以行进路线来检测是否碰撞的方法主要用于检测两个移动的物体或者移动物体与平面是否发生碰撞。

### 课后练习

1. 请写出加速度与速度的关系式。
2. 请写出常见的碰撞检测的方法。
3. 试着写出可以用来代表“雪花”粒子的自定义结构“snow”。

# 第 7 章 进入 3D 世界

## 7.1 初探 DirectX

对于对游戏设计有兴趣的读者而言,“DirectX”是一个相当熟悉的字眼。在本书的后半段内容中,将逐步介绍它在游戏设计上的强大威力。

在这一章里,将介绍有关 DirectX 的概念与用途,然后再介绍可以说是 DirectX 中主角的“DirectDraw”。

### 7.1.1 DirectX SDK 简介

当 Windows 系统成为个人计算机的标准操作系统后,多任务与共通的特性让软件工程师不必再为兼容性的问题伤脑筋,但这些特性的副作用就是“慢”,使得直接存取图像、声音与硬件能力受到很大的限制,甚至让高度的声光效果游戏停留在 MS-DOS 阶段。直到 DirectX 的前身“Game SDK”的出现,程序才可以直接存取硬件,游戏才得以在 Windows 平台上快速发展。

DirectX SDK (DirectX Software Develop Kit) 是微软 (Microsoft) 公司开发的一套主要用于设计多媒体、2D、3D 游戏及程序的 API,其中包含了各类与制作多媒体功能相关的组件 (Component),各组件则提供了许多处理多媒体的接口与方法。下面先来一窥 DirectX 的内容,如表 7-1 所示。

表 7-1

组件名称	用途说明
Direct Graphics	DirectX 绘图引擎,负责把影像绘制到屏幕上
DirectAudio	控制声音设备以及各种音效的处理
DirectInput	处理各种输入设备 (鼠标、键盘、摇杆) 的消息
DirectShow	播放影片与多媒体的方法
DirectPlay	用来建立多人联机的网络功能

利用 DirectX SDK 开发出来的应用程序,必须在安装有“DirectX 客户端”的计算机上才能正常运行。后面的内容将介绍运用 DirectX 设计游戏程序的技巧,读者必须先计算机上完成 DirectX SDK 的安装,可以到微软的网站“<http://www.microsoft.com/directx>”上下载最新版本。

### 7.1.2 DirectX 的特色

看过上一小节的内容之后,您可能会产生疑问:“都是 API 函数,这些不同功能的 API 在 MFC 中都有,为什么还要使用 DirectX 呢?”,看了下面的讲解就会明白。



## 1. 与硬件无关的特性

Windows 的最大优点就是软件与硬件无关。软件工程师不需考虑硬件芯片、制造商或等级,只需要针对 DirectX 来设计,DirectX 会通过硬件驱动程序把程序转换成硬件所属的相关命令。

## 2. 直接存取显存

目前的大部分显示卡都已经内建了 8MB~32MB 的显存,而 DirectDraw 可以直接存取这些显存,并且利用“切换页”的功能,将图形显示的效能发挥得淋漓尽致。

## 3. 支持硬件加速

DirectX 支持硬件的加速功能。当建立 DirectX 对象时,程序会自行查询可使用的硬件,程序设计师不需要担心玩家使用怎样的计算机配备,不论是显示卡、声卡或者是输入外围设备,若程序查询到可使用的硬件,则由硬件 HAL (Hardware Abstraction Layer) 来运行,否则则自动由软件 HEL (Hardware Emulation Layer) 来模拟。

## 4. 网络联机功能

程序设计师可以使用 DirectPlay 轻松开发多人联机游戏,联机的方式可以是局域网、调制解调器及各种通信协议。

综上所述,“DirectX”可视为一种程序设计师与硬件间的接口。程序设计师不需要花费心思去构想如何来编写底层的程序代码与硬件打交道,只须巧妙地运用 DirectX 中的各类组件,便可简单地制作出高效能的游戏程序,这也就是 DirectX 的独到之处。

在了解 DirectX 在游戏程序开发上所扮演的角色之后,接下来从头开始学习使用 DirectX 这套功能强大的 API。

## 7.2 使用 Direct Graphics

由微软 (Microsoft) 公司发展的 DirectX 开发工具函数,经过了多年来不断地改版更新,直到近年的 DirectX 第 8 版之后才表现出它的成熟特点。以往的 DirectX 在使用的架构上非常繁杂,而 DirectX 8.0 的出现让许多繁杂的操作与设定变得更加简单容易,例如 3D 接口的设定,过去需有不下数百行,现在只要一二十行即可解决。它同时也包含了许多更新及更强大的绘图技术、音源效果、网络联机、使用者接口等功能,尤其是在 3D 引擎上的表现,所表现的效果令游戏开发者叹为观止。Direct 9.0 更针对游戏的 3D 化与网络化,提供软件工程师“无痛”的升级环境。

### 7.2.1 介绍 Direct Graphics

在 Direct 7.0 之前,绘图引擎分成 DirectDraw 与 Direct3D,而 Direct 8.0 则针对游戏 3D 化而将 DirectDraw 与 Direct3D 合并,取名为 Direct Graphics,专门用来处理 3D 绘图影像及利用 3D 命令的硬件加速特性来发展更强大的 API 函数。到目前为止,大部分在 Windows 操作系统下运行的 3D 游戏,都是由 Direct3D 所发展而来的。

Direct3D 提供了操作系统与硬件设备之间绝佳的兼容性，它克服了以往开发工具上（例如 OpenGL）一贯的缺点。Direct3D 提供了两种 API 模式，分别如下所示：

### 1. 立即模式

立即模式是属于较低层的 API 函数，不易使用，但是却可以帮助在撰写程序时提供较大的弹性来开发 3D 环境，而且所开发出来的游戏在计算机上运行时也能够提供最好的效率表现。

### 2. 保留模式

保留模式比立即模式容易学习与使用，它架构在立即模式的上一层，但是目前并不支持许多重要的新技术，所以这里不加以讨论。

3D 游戏的构成是由程序代码来建构游戏中的场景、模型、贴图管理、特效及光影效果等技术。通常要表现出这些真实的 3D 效果需要耗费掉相当大的系统资源，如果游戏的架构规划不很明确，往往会表现出令人无法忍受的画面及游戏中较低的运行效率。因此开发 3D 游戏之前，就要考虑其游戏画面品质的呈现与如何节省资源等问题，而 Direct3D 可以在这两者之间做一种良好的抉择，所以 Direct3D 的立即模式运用将是开发游戏时的重心。

Direct Graphics 的出现的确让爱用 DirectDraw 的软件工程师裹足不前，原因在于 DirectDraw 简单易学，在纯 2D 环境的平面绘图上有相当不错的表现。

不过 Direct 8.0 取消 DirectDraw，“强迫”使用 3D 平台来处理 2D 接口。换个角度想，DirectDraw 除了简单外，要做出相当炫的特效还得自己动手写，而在 3D 硬件加速卡如此普及化的今天，运用 3D 功能做出超炫的画面也是轻而易举的，若放着好端端的功能不用，游戏的精彩度早已输在起跑点上了。

本章节中将针对 Direct Graphics 的功能做 2D 平面影像处理，一步步踏入 3D 绘图的领域。

## 7.2.2 如何建立 Direct Graphics 设备

在正式踏入 Direct Graphics 的虚拟世界之前，首先需要建立一个 Direct Graphics 的基本环境，用来作为学习绘图的起始点。在编写程序之前，第一步要做的事情就是规划一个程序框架，以后所有工作都可以在这个框架上运行。

范例 7\_1（见随书光盘）是用 MFC 架构完成的，其中的程序包含了表 7-2 中的内容。

表 7-2

范例 ch7_1	
canvasApp.cpp 与 canvasApp.h	程序进入点与窗口建立程序
canvasFrame.cpp 与 canvasFrame.h	窗口框架程序
myd3d.cpp 与 myd3d.h	Direct Graphics 函数库

在此不打算介绍如何使用 MFC，只就 Direct Graphics 的相关部分进行说明。这里假设读者已经了解 MFC 的基本架构并且掌握了 C++ 的基本语法。主要的 Direct Graphics 的相关程序放在 myd3d.cpp 与 myd3d.h 之中，无论使用 MFC、SDK 或是 BC++，不需要做任何变动便可使用。

使用 Direct Graphics 建立进入点与删除点。

## **canvasFrame.cpp 部分程序代码**

```

1  LRESULT canvasFrame::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
2  {
3      switch( message )
4      {
5          case WM_CREATE :
6              if( !d3dCreate( m_hwnd , 640 , 480 , true ))
7                  PostMessage( WM_CLOSE );
8              return 0 ;
9          case WM_DESTROY :
10             d3dRelease();
11             return 0 ;
12     }
13     return CFrameWnd::WindowProc(message, wParam, lParam);
14 }

```

## **程序说明**

(1) 第 5 ~8 行: 取得窗口建立消息, 调用“d3dCreate”作为 Direct Graphics 建立的进入点; 如果建立失败, 发出消息 WM\_CLOSE 来结束程序。

(2) 第 9 ~11 行: 取得窗口结束消息, 调用“d3dRelease”删除为 Direct Graphics 所建立的相关对象。

接下来看看 d3dCreate 是如何运作的。

## **myd3d.cpp 部分程序代码**

```

1  LPDIRECT3D9          d3d_3D ;
2  LPDIRECT3DDEVICE9    d3d_Device ;

```

LPDIRECT3D9 提供其他 Direct Graphics 界面的来源。

LPDIRECT3DDEVICE9 通知硬件做影像处理的主要设备。

## **myd3d.cpp 部分程序代码**

```

1  BOOL d3dCreate( HWND hwnd , int width , int height , BOOL IsWindow )
2  {
3      //调整屏幕使用区大小
4      if( !d3dSetDisplaySize( hwnd , width , height ))
5          return false ;
6      //建立 Direct Graphics 设备
7      if( !d3dDeviceCreate( hwnd , IsWindow ))
8          return false ;
9      //绘图页清空
10     d3dClear();
11     //结束
12     return true ;
13 }

```

Direct Graphics 的初始化可分为 3 部分。

(1) 调用 d3dSetDisplaySize 调整屏幕使用区大小, 使建立窗口的使用区符合设定的大小。此工作适用于窗口模式, 若游戏是在全屏幕运行时可以不进行, 相关内容请自行查阅 Windows 程序设

计书籍。

(2) 调用 `d3dDeviceCreate` 建立 Direct Graphics。

(3) 调用 `d3dClear` 将绘图页清空。

接着就来动手建立 Direct Graphics。

#### myd3d.cpp 部分程序代码

```

1  BOOL d3dDeviceCreate( HWND hWnd , BOOL IsWindow )
2  {
3  //建立 d3d 主要设备
4  d3d_3D = Direct3DCreate9( D3D_SDK_VERSION );
5  if( !d3d_3D )
6  return false ;
7  //取得目前桌面显示模式
8  D3DDISPLAYMODE d3ddm ;
9  if( d3d_3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT , &d3ddm ) != D3D_OK )
10 return false ;
11 //建立绘图设备
12 D3DPRESENT_PARAMETERS d3dpp ;
13 memset( &d3dpp , 0 , sizeof( d3dpp ) );
14 d3dpp.Windowed = IsWindow ;
15 d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD ;
16 d3dpp.BackBufferFormat = d3ddm.Format ;
17 d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
18 d3dpp.EnableAutoDepthStencil = TRUE;
19 d3dpp.Flags = D3DPRESENTFLAG_LOCKABLE_BACKBUFFER ;
20 if( d3d_3D->CreateDevice( D3DADAPTER_DEFAULT ,
21 D3DDEVTYPE_HAL , hWnd , D3DCREATE_SOFTWARE_VERTEXPROCESSING ,
22 &d3dpp , &d3d_Device ) != D3D_OK )
23 return false ;
24 return true ;
25 }

```

#### 程序说明

(1) 第 3~6 行：调用 `Direct3DCreate9` 来产生 Direct Graphics 的 COM 对象，参数 `D3D_SDK_VERSION` 可以确保标头文件与系统上的 DirectX 版本相符。

(2) 第 7、10 行：利用 `GetAdapterDisplayMode` 取得目前显示模式的信息，窗口模式下一定要进行这一步骤。

(3) 第 12~18 行：定义 `D3DPRESENT_PARAMETERS` 成员，并且设定其内定资料。其成员资料如表 7-3 所示。

表 7-3

Windowed	是否为窗口模式
SwapEffect	换页模式
BackBufferFormat	后缓冲区格式
EnableAutoDepthStencil	是否让 DirectX 自动管理深度缓冲区
AutoDepthStencilFormat	指定深度缓冲区格式
Flags	是否必须锁定后缓冲区

(4) 第 19~22 行: 将 d3dpp 设定完毕之后, 利用 CreateDevice 函数来产生符合目前显示模式的设备。

建立 Direct Graphics 成功后, 下面进行绘图页清空的工作。

#### myd3d.cpp 部分程序代码

```
1 void d3dClear( UINT color )
2 {
3     d3d_Device->Clear( 0, 0, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER, color, 1.0f, 0 );
4 }
```

#### 程序说明

标识符 D3DCLEAR\_TARGET 和 D3DCLEAR\_ZBUFFER 指定绘图页及深度缓冲区清空, 一般而言, 绘图页指定颜色为黑色, 深度缓冲区为 1。

接下来, 释放 Direct Graphics 对象。

#### myd3d.cpp 部分程序代码

```
1 void d3dRelease()
2 {
3     if( d3d_3D ) d3d_3D->Release();
4     if( d3d_Device ) d3d_Device->Release();
5 }
6
```

#### 程序说明

调用 d3dRelease() 可以将所建立的对象由内存释放掉, 前提是已确定建立成功。

最后再来看看一个 Direct Graphics 环境的基本绘图方式。其程序与说明如下所示:

#### canvasFrame.cpp 部分程序代码

```
1 void canvasFrame::Render()
2 {
3     //清空绘图页
4     d3dClear(0);
5     //将绘图页画到屏幕上
6     d3d_Device->Present( NULL, NULL, NULL, NULL );
7 }
```

#### 程序说明

(1) 第 4 行: 将绘图页清空, 参数 0 代表黑色, 可使用 D3DCOLOR\_XRGB 来指定其他颜色。

(2) 第 7 行: 将后缓冲区的资料翻至前景。

#### 运行结果

程序运行结果如图 7-1 所示。



Windows 窗口里产生了一个 D3D 绘图装置

图 7-1

如果把“d3dClear”参数改成“D3DCOLOR\_XRGB(0,0,255)”，程序运行效果如图 7-2 所示。



图 7-2

在了解如何建立一个简单的 Windows 窗口与 Direct Graphics 设备之后，接下来便可以开始在 Direct Graphics 设备上绘制任何需要的图形与画面了。

### 7.2.3 使用 Direct Graphics 取得绘图设备（GDI）

Windows 的绘图设备（GDI）虽然较慢，却也提供了相当多的绘图功能与硬件信息。虽然 Direct Graphics 希望用户遗忘 Windows 的 GDI，不过使用 GDI 却可以更方便地完成工作了。

早期 DirectDraw 时代，只要调用 GetDC 就可以取得绘图页，在 Direct Graphics 里首先要面对的就是如何取得绘图页。范例 ch7\_2 中（见随书光盘）myd3d.h 给出了自定的对象 d3dHdc：

#### myd3d.h 部分程序代码

```
1 class d3dHdc
2 {
3 private :
4     HDC m_hdc ;
5     LPDIRECT3DSURFACES m_Surface ;
6 public :
7     void Release();
8     inline operator HDC(){ return m_hdc ;};
9 public :
10    d3dHdc();
```

```
11 ~d3dHdc();
12 };
```

## 程序说明

d3dHdc 里主要是运用 class 语法方便初始化与删除的工作。  
现在来看看如何取得 HDC。

## myd3d.cpp 部分程序代码

```
1  d3dHdc::d3dHdc()
2  {
3  m_hdc = 0 ;
4  m_Surface = 0 ;
5  //取得设备
6  if( !d3d_Device )
7  return ;
8  if( d3d_Device->GetBackBuffer( 0 , 0 , D3DBACKBUFFER_TYPE_MONO , &m_Surface ) !=
   D3D_OK )
9  return ;
10 m_Surface->GetDC( &m_hdc );
11 }
```

## 程序说明

- (1) 第 8 行：调用 GetBackBuffer 绘图页。
- (2) 第 10 行：调用 GetDC 取得绘图设备。

LPDIRECT3DSURFACE9 就是 DirectDraw 的绘图页 (LPDIRECTDRAWSURFACE)，功能却少了很多，在 Direct Graphics 下并不适合用做 2D 贴图，这里就不另做说明。

GetDC 是把整个绘图页锁住，如果调用 GetDC 失败，请查明建立函数 d3dDeviceCreate 里，结构 D3DPRESENT\_PARAMETERS 的参数 Flags 是否设为可锁定 (D3DPRESENTFLAG\_LOCKABLE\_BACKBUFFER)，如果未设，就无法顺利取得绘图设备。

如果不再使用，一定要把取得的设备释放掉，否则 Direct Graphics 无法在上面绘制任何图型，接下来看看如何释放。

## myd3d.cpp 部分程序代码

```
1  void d3dHdc::Release()
2  {
3  if( m_Surface )
4  {
5  if( m_hdc )
6  m_Surface->ReleaseDC( m_hdc );
7  m_Surface->Release();
8  m_hdc = NULL ;
9  m_Surface = NULL ;
10 }
11 }
```

## 程序说明

- (1) 第 5~6 行：确定已建立 HDC 时，将绘图页调用 Release 释放掉。
- (2) 第 7~9 行：把之前取得的绘图页释放掉，并设为 NULL。

最后，试着用 d3dHdc 对象来打上一些文字。

#### canvasFrame.cpp 部分程序代码

```
1 void canvasFrame::Render()  
2 {  
3     LPCTSTR str = "欢迎来到 Direct Graphics 的世界" ;  
4     //清空  
5     d3dClear();  
6     //用 hdc 打上一些文字  
7     d3dHdc hdc ;  
8     SetTextColor( hdc , RGB( 255 , 255 , 255 ));  
9     SetBkMode( hdc , 1 );  
10    TextOut( hdc , 0 , 0 , str , strlen( str ));  
11    hdc.Release();  
12    //成相  
13    d3d_Device->Present( NULL , NULL , NULL , NULL );  
14 }
```

#### 程序说明

- (1) 第 7 行：使用 d3dHdc，在初始化时取得 Direct Graphics 的 Hdc。
- (2) 第 8~10 行：通过 d3dHdc 对象输入一些文字。
- (3) 第 11 行：最后释放掉它。

#### 运行结果

程序运行结果如图 7-3 所示。



图 7-3

到目前为止已经清楚了 Direct Graphics 是如何与 Windows 沟通的，并学会了如何取得 GDI 设备，从下一节开始，将进入到 Direct Graphics 的重头戏——绘图功能。



## 7.3 使用 Direct Graphics 进行 2D 影像处理

在 DirectDraw 时代，只要调用 BltFast 指定几个贴图的位置，就能轻松地把影像文件贴到画面上，也能调用 Blt 来进行简单的影像处理，绘制成想要的结果。而 DirectX 8.0 出现之后，Direct Graphics 取代了以前 DirectDraw 的工作，是否意味着 2D 时代的结束呢？其实不然，过去的游戏只要贴贴图、画画几何图形或做些简单的动画，就可以上市。但科技日新月异，就算是 2D 的 RPG 游戏，无论内容多么精彩，若没有强大的声光效果支持，也很难满足消费者的胃口。游戏内容的好坏并非程序所能控制，但画面的亮丽可用 Direct Graphics 呈现。

### 7.3.1 Direct Graphics 绘图引擎

所谓绘图引擎（Rendering Engine）在这里指的是实际的绘图机制，将输入的命令运行后的结果显示在屏幕上。

Direct Graphics 绘图引擎的架构如下：

- 坐标转换（Transform）：参考世界（World）、视角（View）及投射（Projection）三种矩阵及剪裁（Viewport）参数，做顶点坐标的转换，最后得出实际屏幕绘制位置。
- 色彩计算（Lighting）：依照目前空间中的发射光源、材质属性、环境光与雾的设定，计算各顶点最后的颜色。
- 平面绘制（Raster）：贴图、基台操作、混色，加上前面两项计算的结果，实际绘制图形到屏幕上。

在 Direct Graphics 架构下，不需要了解其内部运算，更不需要知道计算的方法。本书将指导适当的调用时机，并设定参数，然后就由 Direct Graphics 完成所想要的效果。

### 7.3.2 如何贴影像文件

先来学习如何用 Direct Graphics 来取代 DirectDraw，再来研究如何修改各顶点资料以制作出做不到的效果。

首先将学会如何加载图文件，在范例 ch7\_3（见随书光盘）的 myd3d.h 中可找到对象 d3dTexture：

**myd3d.h 部分程序代码**

```
1  class d3dTexture
2  {
3  private :
4      int      m_Width ;
5      int      m_Height ;
6      LPDIRECT3DTEXTURE9  m_Texture ;
7  public :
8      void BltFast( int x , int y );
9      void BltFast( int l , int t , int r , int b );
10 public :
11  BOOL Create( LPCTSTR file );
```

```

12 void Release();
13 inline operator LPDIRECT3DTEXTURE9(){ return m_Texture ;};
14 public :
15 d3dTexture();
16 ~d3dTexture();
17 };

```

**程序说明**

- (1) 第 4~5 行: 记录影像的长宽。  
 (2) 第 6 行: Direct Graphics 材质的接口。

**myd3d.cpp 部分程序代码**

```

1  BOOL d3dTexture::Create( LPCTSTR file )
2  {
3      D3DXIMAGE_INFO in ;
4      memset( &in , 0 , sizeof( in ) );
5      //初始化
6      Release();
7      //加载
8      D3DXCreateTextureFromFileEx( d3d_Device ,
9      file , D3DX_DEFAULT , D3DX_DEFAULT ,
10     0 , 0 , D3DFMT_UNKNOWN , D3DPOOL_MANAGED ,
11     D3DX_DEFAULT ,
12     D3DX_DEFAULT , 0 , &in , NULL , &m_Texture );
13     if( m_Texture == NULL )
14         return false ;
15     //取得资料
16     m_Width = in.Width ;
17     m_Height = in.Height ;
18     return true ;
19 }

```

**程序说明**

(1) 第 8~12 行: 试着读取影像文件, 并将其值定义给 &m\_Texture 材质对象, 将资料存放在结构 D3DXIMAGE\_INFO 上。

(2) 第 16~17 行: 由结构 D3DXIMAGE\_INFO 上取得影像的长宽。

接下来, 将材质“绑”在各顶点上, 并“告诉”平面 Direct Graphics 要如何画到屏幕上。首先需要定义 2D 绘图常用的顶点资料, 用过 Direct 7.0 版的 Direct3D 的使用者一定会对下面的定义很熟悉。

**myd3d.h 部分程序代码**

```

1  const DWORD D3DFVF_TLVERTEX = (D3DFVF_XYZRHW | D3DFVF_DIFFUSE |
2      D3DFVF_SPECULAR | D3DFVF_TEX1 );
3  typedef struct _D3DTLVERTEX
4  {
5      float x , y , z , rhw ;
6      D3DCOLOR diffuse , specular;
7      float tu, tv;
8  }D3DTLVERTEX ;

```

## 程序说明

- (1) 第 1~2 行：告诉 Direct Graphics 画面已转换、已打光，含一组材质坐标。
- (2) 第 2~7 行：定义顶点的数据结构。

这是标准的已上色已打光的顶点结，先回忆学过的 Direct Graphics 的绘图引擎，在三大模块中，只用了平面绘制（Raster），转换工作和颜色都以计算完成，只要将结果告诉 Direct Graphics 就可以了，这也是最接近屏幕输出的结果。

利用这个结构，可试着设定各顶点的位置、颜色、材质坐标，再由 Direct Graphics 画出。

## myd3d.h 部分程序代码

```
1 void d3dTexture::BlitFast(int l, int t, int r, int b)
2 {
3     D3DTLVERTEX v[4];
4     //顶点的结构
5     memset(v, 0, sizeof(v));
6     v[0].x = v[3].x = (float)(l);
7     v[1].x = v[2].x = (float)(r);
8     v[0].y = v[1].y = (float)(t);
9     v[2].y = v[3].y = (float)(b);
10    v[0].rhw = v[1].rhw = v[2].rhw = v[3].rhw =
11    v[0].z = v[1].z = v[2].z = v[3].z = 0.5f;
12    v[0].diffuse = v[1].diffuse = v[2].diffuse = v[3].diffuse = -1;
13    v[1].tu = v[2].tu = 1.0f;
14    v[2].tv = v[3].tv = 1.0f;
15    //设定绘图模式
16    d3d_Device->SetTexture(0, m_Texture);
17    d3d_Device->SetFVF(D3DFVF_TLVERTEX);
18    d3d_Device->DrawPrimitiveUP(D3DPT_TRIANGLEFAN, 2, (LPVOID)v,
19                                sizeof(D3DTLVERTEX));
20 }
```

## 程序说明

- (1) 第 6~11 行：设定屏幕 4 个顶点的位置及深度缓冲区（ZBuffer）。
- (2) 第 12 行：设定顶点颜色。
- (3) 第 13~14 行：设定贴图坐标。
- (4) 第 16 行：设定贴图材质。
- (5) 第 17 行：告诉 Direct Graphics 这些点的格式。
- (6) 第 18 行：以三角扇形（D3DPT\_TRIANGLEFAN）方式由 Direct Graphics 画出这些顶点。

DrawPrimitiveUP 声明如表 7-4 所示。

表 7-4

DrawPrimitiveUP 声明	
D3DPRIMITIVETYPE PrimitiveType	三角形的样式
UINT PrimitiveCount	三角形的数量
const void *pVertexStreamZeroData	顶点的资料
UINT VertexStreamZeroStride	顶点结构的大小

Direct Graphics 可绘制的三角形方法分为以下 3 种。

### (1) 三角形清单 (Triangle lists)

这是最为简单且广泛被使用的三角形清单顶点构成法。当要构成一个基本的三角形时, 给每一个三角形定义三个顶点, 如果要绘制 15 个三角形的话, 那就要定义 45 个顶点, 依次类推。如图 7-4 所示。

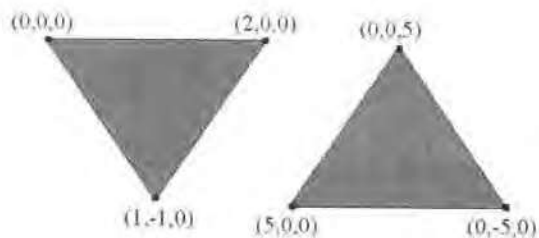


图 7-4

### (2) 三角形带 (Triangle strips)

在三角形带中, 第一个三角形由  $v_1$ 、 $v_2$ 、 $v_3$  构成。接下来 D3D 使用第一个三角形中的后两个顶点与一个新的顶点来构成另外一个三角形。依次类推, D3D 使用第二个三角形的后两个顶点与一个新顶点来构成第三个三角形。所以, 三角形带的构成法是利用前一个三角形的后两个顶点与另一个新顶点来构成一个新的三角形。如图 7-5 所示。

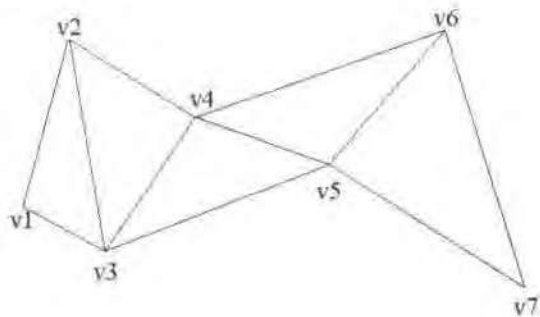


图 7-5

### (3) 三角形扇 (Triangle fans)

D3D 设备亦可以将顶点解读为三角形扇的类型。它以一个顶点为基准点, 然后再以其他的顶点来构成数个三角形, 形成一个如同扇状的几何图形。如图 7-6 所示。

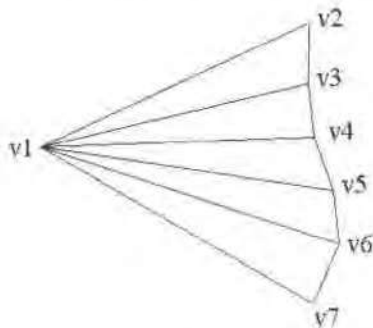


图 7-6

在前面的函数中，使用两个三角形构成三角扇形来绘制四边形，各顶点的顺序如图 7-7 所示。

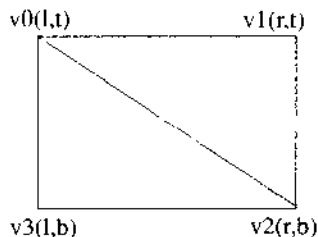


图 7-7

由于这是 2D 平面绘图，顶点的 Z 及 rhw 只能设在 0~1 之间，这里统一设为 0.5。

最后考虑贴图坐标 tu 与 tv。首先，在顶点的结构中定义了两个浮点数的资料变量，点阵影像图所代表的坐标单位是一个 2 维的坐标系统，所以就利用 u 代表 x 轴的坐标单位，而 v 则是代表 y 轴的坐标单位。在 Direct Graphics 的环境中，u 值的最大值为 1.0f、最小值为 0.0f，如图 7-8 所示。

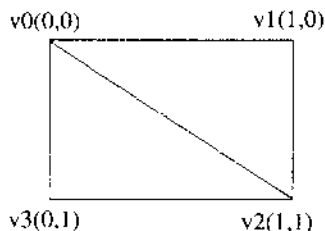


图 7-8

为了更简单地计算绘图位置，另外编写了下面的函数，只要输入左上角的点，就可以画出整张图。

## myd3d.h 部分程序代码

```
1 void d3dTexture::BltFast(int x, int y)
2 {
3     BltFast( x , y , x + m_Width , y + m_Height );
4 }
```

接下来就试着把图画到屏幕上。在 canvasFrame.cpp 里声明 d3dTexture 对象，用来加载两张图。

```
1 d3dTexture a_Bk ;
2 d3dTexture a_Role ;
```

接下来在窗口建立消息上加载两张 tga 的图文件，当然也可以使用 bmp 或 jpg 的图文件。

```
1 case WM_CREATE :
2     if( !d3dCreate( m_hWnd , 640 , 480 , true ) )
3         return PostMessage( WM_CLOSE );
4     a_Bk.Create( "背景.tga" );
5     a_Role.Create( "娃娃.tga" );
6     return 0 ;
```

最后在绘图函数中画出这两张图。

## canvasFrame.cpp 部分程序代码

```
1 void canvasFrame::Render()
```

```
2 {  
3 // d3dTexture bk ;  
4 //清空  
5 d3dClear();  
6 //开始绘制  
7 d3d_Device->BeginScene();  
8 d3d_Device->SetRenderState( D3DRS_CULLMODE , D3DCULL_NONE );  
9 d3d_Device->SetRenderState( D3DRS_ZENABLE , D3DZB_FALSE );  
10 a_Bk.BlitFast( 0 , 0 );  
11 a_Role.BlitFast( 0 , 0 );  
12 d3d_Device->EndScene();  
13 //成像  
14 d3d_Device->Present( NULL , NULL , NULL , NULL );  
15 }
```

#### 程序说明

- (1) 第 7 行: 通知 Direct Graphics 将开始绘制。
- (2) 第 8 行: 设定顶点的顺序为正反面都画。
- (3) 第 9 行: 这是 2D 绘图, 所以关掉深度缓冲区。
- (4) 第 10~11 行: 将两张图画出来。
- (5) 第 12 行: 通知 Direct Graphics 绘制结束。
- (6) 第 14 行: 最后显示在屏幕上。

#### 运行结果

程序运行结果如图 7-9 所示。



图 7-9

在 Direct Graphics 进行绘图之前, 一定要调用 BeginScene 通知其准备开始绘制。同样, 当绘制结束时, 也要调用 EndScene 通知 Direct Graphics 绘图已完成。和 GDI 的绘图原理一样, 所有的绘图动画工作都要在这两个函数之间运行。

三角形的顶点有一定的顺序性, Direct Graphics 以顶点的顺序来决定面的方向, 若面的方向背着使用者, 就不绘制它。而在 2D 绘图中, 每个面都要看得到。

在调用 BltFast 时,也可以直接指定 4 个边的位置来进行放大、缩小及镜面映射,例如前面范例中把

```
a_Role.BltFast( 0 , 0 );
```

改成

```
a_Role.BltFast( 300 , 300 , 100 , 100 );
```

就可以达到镜面反射及缩小的功能,如图 7-10 所示。



图 7-10

既然可以用更改顶点的方式来完成放大、缩小与镜面映射,当然也可用同样的方式进行旋转。在范例 ch7\_4 中(见随书光盘)对 BltFast 做些修改,以计算旋转的新位置,程序代码如下。

myd3d.cpp

```
1 void didTexture::BltFast(int l , int t , int r , int b , float a )
2 {
3     D3DTLVERTEX v[4] ;
4     //定点的结构
5     memset( v , 0 , sizeof( v ) );
6     //点位置
7     //不旋转
8     if( a == 0.0f )
9     {
10        v[0].x = v[3].x = (float)(l) ;
11        v[1].x = v[2].x = (float)(r);
12        v[0].y = v[1].y = (float)(t);
13        v[2].y = v[3].y = (float)(b);
14    }else //旋转
15    {
16        float ox , oy ;
17        float in ;
18        float s , c ;
```

```

19 //角度转弧度
20 in = a * 3.1415926f / 180.0f ;
21 //原点
22 ox = (float)( r + l ) / 2 ;
23 oy = (float)( b + t ) / 2 ;
24 //相对位置
25 l -= (int)ox ;
26 t -= (int)oy ;
27 r -= (int)ox ;
28 b -= (int)oy ;
29 //计算新位置
30 s = sinf( in );
31 c = cosf( in );
32
33 v[0].x = c * l + s * t + ox ;
34 v[0].y = -s * l + c * t + oy ;
35
36 v[1].x = c * r + s * t + ox ;
37 v[1].y = -s * r + c * t + oy ;
38
39 v[2].x = c * r + s * b + ox ;
40 v[2].y = -s * r + c * b + oy ;
41
42 v[3].x = c * l + s * b + ox ;
43 v[3].y = -s * l + c * b + oy ;
44 }
45 //Z
46 v[0].rhw = v[1].rhw = v[2].rhw = v[3].rhw =
47 v[0].z = v[1].z = v[2].z = v[3].z = 0.5f ;
48 //颜色
49 v[0].diffuse = v[1].diffuse = v[2].diffuse = v[3].diffuse = -1 ;
50 //材质
51 v[1].tu = v[2].tu = 1.0f ;
52 v[2].tv = v[3].tv = 1.0f ;
53 //设定绘图模式
54
55 d3d_Device->SetTexture( 0 , m_Texture );
56 d3d_Device->SetFVF( D3DFVF_TLVERTEX );
57 d3d_Device->DrawPrimitiveUP( D3DPT_TRIANGLEFAN , 2 , (LPVOID)v ,
    sizeof( D3DTLVERTEX ) );
58
59 }

```

#### 程序说明

第 16~40 行用来计算以绘制的中心点为轴旋转后 4 个顶点的新位置，计算流程如下：

- (1) 先计算绘制点的中心位置 (22~23 行)；
- (2) 以中心点为轴，计算四个边的相对距离 (25~28 行)；
- (3) 对 4 个边进行二阶矩阵转换，二阶旋转矩阵如下，而  $(Tx, Ty)$  代表中心点坐标：



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta & Tx \\ -\sin \theta & \cos \theta & Ty \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

(4) 二阶矩阵转换后出来的 4 个新顶点即是旋转后的位置。

接着来实际操作，在函数 `Render` 里绘制两个娃娃，一个不旋转，一个顺时针旋转  $45^\circ$ 。可以在 `canvasFrame.cpp` 找到以下两个绘制点。

```
a_Role.BitFast( 100, 100, 250, 250, 0 );
a_Role.BitFast( 400, 100, 550, 250, 45.0f );
```

## 运行结果

程序运行结果如图 7-11 所示。



图 7-11

以上介绍了如何贴图及如何更改顶点做一些简单的影像处理。到此为止，用 `Direct Graphics` 完全取代了 `DirectDraw` 的工作，读者是否开始感受到 `Direct Graphics` 强大的威力呢？

贴图是最基本的工作，也是游戏最表面的工作，如果还不熟悉其中的架构，建议再多读几遍或尝试着动手更改里面的参数，相信可以更加了解它。

## 7.4 Direct Graphics 的颜色操作

在典型的 2D 游戏中，软件工程师会使用颜色键 (`ColorKey`) 去背，或是运用 `Alpha` 来处理透明等级。仅计算公式可能就不是新手所能学会的，更不用说在硬件限制下，常会为画面与效率如何平衡而头痛不已了。

在 `Direct Graphics` 的架构下，上面所述的痛苦日子都将成为回忆。就算是新手，只要了解 `Direct Graphics` 的颜色操作，再进行几个小时的练习，就可以超过有数十年经验的设计者。

### 7.4.1 Direct Graphics 颜色操作流程

首先来看看 Direct Graphics 的颜色操作流程，如图 7-12 所示。

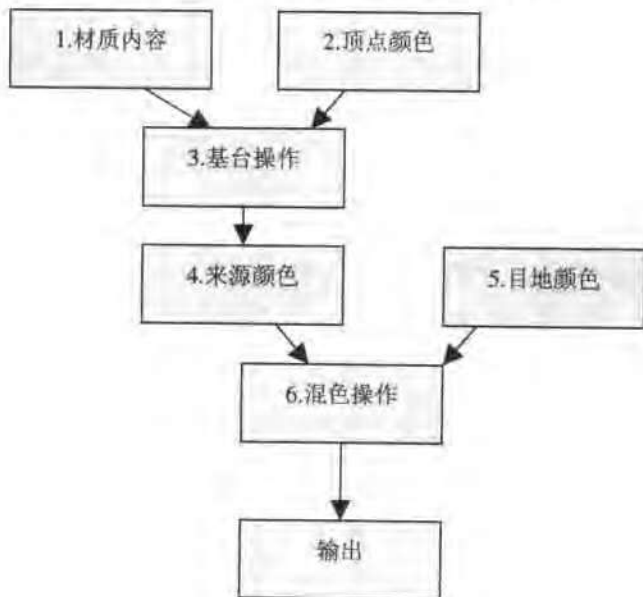


图 7-12

- 材质内容：这里所讲的材质，即所谓的图片文件。一般而言，材质可分为红、绿、蓝及混色值（Alpha）。
- 顶点颜色：顶点的色彩值。
- 基台操作：可分为颜色（Color）与混色（Alpha）两种操作，将这两个值混合后成为新的输出色彩。
- 来源颜色：这里指的是经由基台操作所得到的颜色，作为后缓冲区输出前的色彩。
- 目的颜色：已绘制到后缓冲区的色彩。
- 混色操作：对来源色彩与目的色彩的操作，得到的结果就是最后输出的结果。

### 7.4.2 混色操作

了解了 Direct Graphics 的色彩操作后，下面就来实际操作一遍，首先来学习如何将图片去背。

在 DirectDraw 时代，只要直接设定颜色就可以很简单地去背了，Direct Graphics 没有颜色键的设定，是利用混色操作来进行的。打开范例 ch7\_5（见随书光盘），可以在函数 Render 中找到以下程序代码：

canvasFrame.cpp 部分程序代码

```
1 void canvasFrame::Render()  
2 {  
3     //清空  
4     d3dClear();  
5     //开始绘制
```

```

6  d3d_Device->BeginScene();
7  d3d_Device->SetRenderState( D3DRS_CULLMODE , D3DCULL_NONE );
8  d3d_Device->SetRenderState( D3DRS_ZENABLE , D3DZB_FALSE );
9  d3d_Device->SetRenderState( D3DRS_SHADEMODE , D3DSHADE_FLAT );
10 d3d_Device->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
11 a_Bk.BltFast( 0 , 0 );
12 d3d_Device->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_SRCALPHA );
13 d3d_Device->SetRenderState( D3DRS_DESTBLEND , D3DBLEND_INVSRCALPHA );
14 d3d_Device->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
15 a_Role.BltFast( 0 , 0 );
16 d3d_Device->EndScene();
17 //成像
18 d3d_Device->Present( NULL , NULL , NULL , NULL );
19 }

```

### 程序说明

- (1) 第 10~14 行: 混色操作开启与关闭。
- (2) 第 12 行: 混色操作来源颜色值设定。
- (3) 第 13 行: 混色操作目的颜色值设定。

### 运行结果

程序运行结果如图 7-13 所示。



图 7-13

上面的程序代码就是简单的范例, 首先使用函数 `HRESULT SetRenderState (D3DRENDERSTATETYPE State, DWORD Value)` 来设定混色操作, 参数 `State` 代表设定项目, `Value` 代表方式。

背景图只要简单地把颜色涂上去可以, 所以调用函数 `SetRenderState (D3DRS_ALPHABLENDENABLE, FALSE)` 将混色计算关闭。第 14 行画娃娃图要用到 `tga` 档里面的 `alpha` 值来当混色来源, 所以再把混色计算打开, 标识符 `D3DRS_ALPHABLENDENABLE` 就是负责混色开关设定的。

第 12~13 行中设定指定的计算模式, 标识符 `D3DRS_SRCBLEND` 设定来源值, 标志符 `D3DRS_DESTBLEND` 代表目的值, 常用的计算参数如下。

- D3DBLEND\_ZERO (0, 0, 0, 0) (全黑)
- D3DBLEND\_ONE (1, 1, 1, 1) (全亮)
- D3DBLEND\_SRCCOLOR (Rs, Gs, Bs, As) (来源颜色)
- D3DBLEND\_INVSRCOLOR (1-Rs, 1-Gs, 1-Bs, 1-As) (反相目的颜色)
- D3DBLEND\_SRCALPHA (As, As, As, As) (来源 Alpha)
- D3DBLEND\_INVSRCALPHA (1-As, 1-As, 1-As, 1-As) (反相来源混色)
- D3DBLEND\_DESTALPHA (Ad, Ad, Ad, Ad) (目的混色)
- D3DBLEND\_INVDESTALPHA (1-Ad, 1-Ad, 1-Ad, 1-Ad) (反相目的混色)
- D3DBLEND\_DESTCOLOR (Rd, Gd, Bd, Ad) (目的颜色)
- D3DBLEND\_INVDESTCOLOR (1-Rd, 1-Gd, 1-Bd, 1-Ad) (反相目的颜色)

表 7-5 列出了一些常用的计算方式。

表 7-5

结 果	来 源 设 定	目 的 设 定
去背	D3DBLEND_SRCCOLOR	D3DBLEND_INVSRCOLOR
打亮与光栅	D3DBLEND_ONE	D3DBLEND_ONE
影子	D3DBLEND_ZERO	D3DBLEND_INVSRCOLOR

例如将来源与目的的设置

```
1 d3d_Device->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_SRCALPHA );
2 d3d_Device->SetRenderState( D3DRS_DESTBLEND , D3DBLEND_INVSRCALPHA );
```

改为

```
1 d3d_Device->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_ONE );
2 d3d_Device->SetRenderState( D3DRS_DESTBLEND , D3DBLEND_ONE );
```

结果将变成图 7-14 所示的效果。



图 7-14

## 7.4.3 材质基台操作

在 Direct Graphics 的环境中，将使用超过一个以上的材质基台来进行多重材质的混合运算，而材质基台的意义在于它会采用两种色彩（即 RGB 与 Alpha 值），并且将它们混合在一起。如图 7-15 所示。



图 7-15

还记得前面定义的顶点结构 D3DTLVERTEX 吗？其中的 diffuse 代表扩散色，specular 代表反射色。现在就来运用它，参看范例 ch7\_6（见随书光盘），在对象 d3dTexture 中，BltFast 新增了一个可以用来设定扩散色的参数，并且设定到 4 个顶点中。可以在 myd3d.cpp 中看到以下程序代码。

```
v[0].diffuse = v[1].diffuse = v[2].diffuse = v[3].diffuse = diffuse ;
```

接下来，在 Render 函数中实际使用它。

**canvasFrame.cpp 部分程序代码**

```
1
2 void canvasFrame::Render()
3 {
4     //清空
5     d3dDevice->Clear();
6     //开始绘制
7     d3dDevice->BeginScene();
8     d3dDevice->SetRenderState( D3DRS_CULLMODE , D3DCULL_NONE );
9     d3dDevice->SetRenderState( D3DRS_ZENABLE , D3DZB_FALSE );
10    d3dDevice->SetRenderState( D3DRS_SHADEMODE , D3DSHADE_FLAT );
11    d3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
12    a_Bk.BltFast( 0 , 0 );
13    //开混色
14    d3dDevice->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_SRCALPHA );
15    d3dDevice->SetRenderState( D3DRS_DESTBLEND , D3DBLEND_INVSRCALPHA );
16    d3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
17    //设定色彩与混色操作来源
18    d3dDevice->SetTextureStageState( 0 , D3DTSS_ALPHAARG1 , D3DTA_TEXTURE );
19    d3dDevice->SetTextureStageState( 0 , D3DTSS_ALPHAARG2 , D3DTA_DIFFUSE );
```

```

20     d3d_Device->SetTextureStageState( 0 , D3DTSS_COLORARG1 , D3DTA_TEXTURE );
21     d3d_Device->SetTextureStageState( 0 , D3DTSS_COLORARG2 , D3DTA_DIFFUSE );
22     // 顶点混色值一律为 0.5 (128)
23     // 来源 ALPHA 值两者相乘, 颜色用基台 1
24     d3d_Device->SetTextureStageState( 0 , D3DTSS_COLOROP , D3DTOP_SELECTARG1 );
25     d3d_Device->SetTextureStageState( 0 , D3DTSS_ALPHAOP , D3DTOP_MODULATE );
26     a_Role.BltFast( 0 , 0 , 200 , 200 , D3DCOLOR_ARGB( 128 , 255 , 255 , 255 ) );
27     // 来源 ALPHA 值基台 1, 颜色用基台 2
28     d3d_Device->SetTextureStageState( 0 , D3DTSS_COLOROP , D3DTOP_SELECTARG2 );
29     d3d_Device->SetTextureStageState( 0 , D3DTSS_ALPHAOP , D3DTOP_SELECTARG1 );
30     a_Role.BltFast( 200 , 0 , 400 , 200 , D3DCOLOR_ARGB( 128 , 255 , 255 , 255 ) );
31     // 来源 ALPHA 值基台 1, 颜色用基台相乘, 顶点设红色
32     d3d_Device->SetTextureStageState( 0 , D3DTSS_COLOROP , D3DTOP_MODULATE );
33     d3d_Device->SetTextureStageState( 0 , D3DTSS_ALPHAOP , D3DTOP_SELECTARG1 );
34     a_Role.BltFast( 000 , 200 , 200 , 400 , D3DCOLOR_ARGB( 128 , 255 , 0 , 0 ) );
35     // 来源 ALPHA 值基台相乘, 颜色用基台相乘, 顶点设黄色
36     d3d_Device->SetTextureStageState( 0 , D3DTSS_COLOROP , D3DTOP_MODULATE );
37     d3d_Device->SetTextureStageState( 0 , D3DTSS_ALPHAOP , D3DTOP_MODULATE );
38     a_Role.BltFast( 200 , 200 , 400 , 400 , D3DCOLOR_ARGB( 128 , 0 , 255 , 255 ) );
39
40     d3d_Device->EndScene();
41     // 成像
42     d3d_Device->Present( NULL , NULL , NULL , NULL );
43
44 )

```

### 运行结果

程序运行结果如图 7-16 所示。



图 7-16

### 程序说明

(1) 第 18~21 行, 设定基台来源。混色与颜色来源 1 通常设为材质色, 来源 2 设为顶点扩散



- D3DTOP\_BLENDCURRENTALPHA: 使用前一个材质 Stage 的 Alpha 值来进行混合运算。
- D3DTA\_DIFFUSE: 告诉 D3D 使用扩散色彩作为输入。
- D3DTA\_SPECULAR: 告诉 D3D 使用反射色彩作为输入。
- D3DTA\_TEXTURE: 告诉 D3D 使用材质作为输入。
- D3DTA\_CURRENT: 告诉 D3D 使用前一个材质 Stage 的输出作为输入。
- D3DTA\_TEMP: 有些显示卡具有色彩资料暂存区, 则可以对该暂存区进行存取。
- D3DTA\_ALPHAREPLICATE: 告诉 D3D 将 Alpha 值复制到所有的 RGB 值上, 以形成一种灰阶的效果。
- D3DTA\_COMPLEMENT: 告诉 D3D 将自变量反转。

## 课后重点整理

- DirectX SDK (DirectX Software Develop Kit) 是微软公司开发的一套主要用于设计多媒体、2D、3D 游戏及程序的 API, 其中包含了各类与制作多媒体功能相关的组件 (Component)。
- 在 Direct 7.0 之前, 绘图引擎分成 DirectDraw 与 Direct3D, 而 Direct 8.0 则针对游戏 3D 化而将 DirectDraw 与 Direct3D 合并, 取名为 Direct Graphics, 专门用于处理 3D 绘图影像及利用 3D 命令的硬件加速特性来发展更强大的 API 函数。
- Direct3D 提供了两种 API 模式: 立即模式、保留模式。
- 在 DirectDraw 时代, 只要调用 BltFast 指定几个贴图的位置就能轻松地把影像文件贴到画面上, 也能调用 Blt 来进行简单的影像处理, 绘制成想要的结果。
- 所谓绘图引擎 (Rendering Engine), 指的是实际的绘图机制, 将输入的命令运行后的结果显示在屏幕上。
- 在 Direct Graphics 环境中, 使用 “SetTextureStageState” 的函数命令来混合材质可以创造出许多虚幻又真实的效果。

### 课后练习

1. 简述 DirectX 的特色。
2. Direct3D 提供了哪两种 API 模式? 试简述之。
3. 试简述 Direct Graphics 绘图引擎的架构。
4. 请说明 Direct Graphics 的颜色操作流程。
5. 试解释 SetTextureStageState 函数命令中各参数所代表的意义。

```
HRESULT SetTextureStageState(
    DWORD Stage,
    D3DTEXTURESTAGESTATETYPE Type,
    DWORD Value
);
```

6. 延续上例, 请分别举出 3 种 “Type” 及 “Value” 这两个参数所能够设定的值。





# 第 8 章 Direct Graphics 3D 的奇幻世界

## 8.1 迷人的 3D 魅力

上一章节中介绍了 Direct Graphics 的平面绘制 (Raster) 模块, 本章将介绍顶点坐标到屏幕坐标的坐标转换 (Transform) 模块, 以及由光线、材质算出顶点颜色的色彩计算 (Lighting) 模块。

### 8.1.1 三维空间概念

3D 指被描述或显示的对象有宽度、高度和深度三个测量维度, 又称为三维。由于 3D 计算机绘图最后只能在屏幕上呈现, Direct Graphics 绘图就像是拿着相机照苹果所洗出来的相片。

如果让相片里出现苹果, 当然一定要有苹果、相机和底片。先将苹果放在任何一个地方, 如桌子或电视机上, 再将相机对准苹果, 装好底片, 在适当的光线下按下快门, 然后等底片洗出来即可。

回顾上一章中提到的 Direct Graphics 的三大模块, 即由坐标转换 (Transform) 画出苹果的“型”, 经过色彩计算 (Lighting) 决定苹果的颜色, 由平面绘制 (Raster) 将照片洗出来。

### 8.1.2 模型与顶点

在现实生活中看得到的任何物品都可称为模型, 而 3D 绘图中所说的模型是用数据表示成顶点的集合, 可以是有具体形态的物体 (如书本、计算机) 或无具体形态的动态 (如浪花、光线) 等, 这些顶点都可以存储在文件中。

在上一节的例子里, 苹果代表模型, 广义的定义是各种几何图型在空间中的位置, 这些位置被称为顶点 (Vertex)。两点形成一条直线 (List), 三点形成一个平面 (Triangle, 又称三角形), 许多个平面就可以构成任何对象, 例如, 四边形可用两个三角形构成, 正立方体可用 12 个。用越多的三角形构成对象, 自然就越接近真实。

关于顶点的使用方法, 在上一章中已经简单地介绍了利用上色和打光绘图, 而 Direct Graphics 使用的顶点当然不仅如此, 下面就按次序列在表 8-1 中, 可以根据程序需要而进行设定。

表 8-1

绘图参数	名称	变量类型	说明
D3DFVF_XYZ	未转换的顶点	3 个浮点数 (float x, y, z)	模型中未转换的各个顶点的原始坐标
D3DFVF_XYZRHW	已转换的顶点	4 个浮点数 (float x, y, z, rhw)	模型中已转换为屏幕坐标的顶点位置, z 值在 0-1 之间, rhw 是 1/z
D3DFVF_XYZB1 到 D3DFVF_XYZB5	混合加权信息参数	1~5 个浮点数	用于多重矩阵顶点的混合基台操作混色 Alpha 值
D3DFVF_NORMAL	顶点法向量	3 个浮点数 (float nx, ny, nz)	计算光的强弱

(续表)

绘图参数	名称	变量类型	说明
D3DFVF_DIFFUSE	扩散颜色	1 个无符号整数 (DWORD diffuse)	该顶点用 RGBA 格式表示的扩散色
D3DFVF_SPECULAR	反射颜色	1 个无符号整数 (DWORD specular)	格式与扩散颜色相同
D3DFVR_TXT0 到 D3DFVF_TXT8	材质坐标	2 个以上的浮点数 (float tu, fv)	表明 Direct Graphics 顶点信息参数内有多少材质坐标

表 8-2 是常用的顶点组合。

表 8-2

结构名称	顶点用途	绘图参数
typedef struct D3DVERTEX { float x, y, z //位置 float nx, ny, nz //法向量 float tu, tv //材质坐标 }D3DVERTEX;	未转换未上色, 任何 3D 模型都很适用	D3DFVF_XYZ   D3DFVF_NORMAL   D3DFVR_TXT1
typedef struct D3DLVERTEX { float x, y, z //位置 DWORD diffuse //扩散色 DWORD specular //反射色 float tu, tv //材质坐标 }D3DLVERTEX;	未转换已打光, 不考虑光源影响或已 计算完成顶点颜色时适用	D3DFVF_XYZ   D3DFVF_DIFFUSE   D3DFVF_SPECULAR   D3DFVR_TXT1
typedef struct D3DTLVERTEX { float x, y, z //屏幕位置 float rhw //深度坐标 DWORD diffuse //扩散色 DWORD specular //反射色 float tu, tv //材质坐标 }D3DTLVERTEX;	已转换已打光, 最接近屏幕绘图的顶 点	D3DFVF_XYZRHW   D3DFVF_DIFFUSE   D3DFVF_SPECULAR   D3DFVF_TEX1

大多数类型都是用 3D Studio Max 建造的, 而利用程序直接建模型的很少。有关 Direct Graphics 的顶点结构, 只要先了解就可以了。我们将直接从 X 模型文件来画模型, 关于转换和打光 (D3DTLVERTEX) 的顶点在上一章中已经详细介绍了, 后面章节将使用实际模型来一一介绍顶点坐标的转换方法 (坐标转换模块) 与顶点颜色的设定方法 (色彩计算模块)。

## 8.1.3 3D 世界的环境描述

构成一个 3D 虚拟环境的最基本原则是: 必须有一个 3D 空间的环境坐标系, 这种环境坐标系分为两大类, 如下所示。

## 1. 右手 3D 坐标系

右手 3D 坐标系是以一般的卡森 2D 坐标系再加上一个“指向”使用者的方向轴构成的坐标系。如图 8-1 所示。

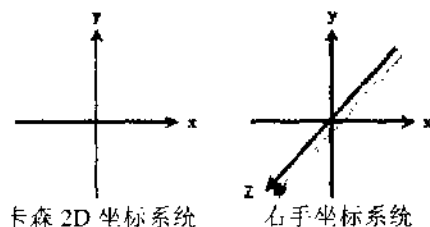


图 8-1

## 2. 左手 3D 坐标系

左手 3D 坐标系是由一般的卡森 2D 坐标系再加上一个“远离”使用者的方向轴所构成的坐标系。如图 8-2 所示。

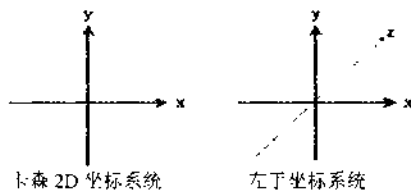


图 8-2

Direct Graphics 使用的是左手坐标系，所以后面范例程序或文本里涉及的 Z 轴方向，就是远离使用者的一个方向轴。

另一个要注意的是顶点的排列方式，一个空间中基本上有两个面向，顶点的顺序直接影响到该面是否可见，也就是 Direct Graphics 判断是否需要在屏幕上绘制。在左手坐标系下，顶点的可见面排列方式为顺时针，如图 8-3 所示，右手坐标系正好完全相反。

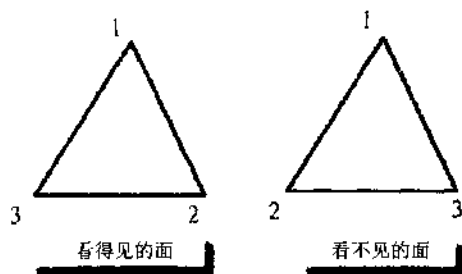


图 8-3

只要调用 `SetRenderStat(D3DRS_CULLMODE, D3DCULL_CCW)` 就可以设定为顺时针。准确地说，顶点顺序是逆时针的面将被挖掉 (Cull)，也可以输入参数 `D3DCULL_NONE`，并设定为无论如何都看得到该面。或是 `D3DCULL_CW` 将顺时针的面挖去，只看得到逆时针的面。

## 8.1.4 顶点颜色的计算方法

虽然在场景中可以直接指定几何模型顶点颜色，在这种情况下并不需要任何光源，但是如果加上几盏灯光，再通过表面材质方向，可以得到更真实的 3D 场景，Direct Graphics 通过设定光源与材质，得到最终顶点的扩散色（diffuse）与反射色（specular）。

要用 Direct Graphics 计算光，必需要提供以下信息条件。

- (1) 顶点法向量（Normal）。
- (2) 环境光或发射光（Light）。
- (3) 模型表面材质（Material）。

Direct Graphics 计算光的方法如下。

- (1) 先换算距离的比例值：（ $d=Range$ ）
- (2) 计算强弱因子：

$$F = Attenuation0 + \\ Attenuation1 * d + \\ Attenuation2 * (d^2)$$

- (3) 该点受到的光强度为  $F * Light$ 。

(4) 根据顶点法向量计算该点的受光强度，最后再根据表面材质（Material）的特性，计算出顶点的扩散色（diffuse）与反射色（specular），填入顶点中。

## 8.1.5 加载一个 X 文件的模型

Direct Graphics 的工具函数库支持 X 文件，现在就来实际创建一个记录模型的对象。打开范例 ch8\_1（见随书光盘）。

### myd3d.h 部分程序代码

```
1  class d3dXFileData
2  {
3  private :
4      DWORD          m_Num ;           //模型数
5      LPD3DXMESH      m_Mesh ;         //模型信息
6      D3DMATERIAL9    *m_Material ;    //表面材质
7      d3dTexture      *m_Texture ;     //材质图文件
8  public :
9      void Init();
10     void Release();
11     void Draw() ;
12     BOOL Create( LPCTSTR file );
13 public :
14     d3dXFileData(){Init();};
15     ~d3dXFileData(){Release();};
16 };
```

### 程序说明

- (1) 第 5 行：LPD3DXMESH 记录所有模型的顶点信息。

(2) 第 6 行: D3DMATERIAL9 为表面材质 (Material), 这里所说的材质和图文件材质 (Texture) 不一样, 这里指的是设定该模型表层的颜色与对“光”的反射度, 计算扩散色 (diffuse) 与反射色 (specular)。

(3) 第 7 行: 贴图用的材质文件。

#### myd3d.cpp 部分程序代码

```

1  BOOL d3dxFileData::Create( LPCWSTR file )
2  {
3      DWORD i ;
4      LPD3DXBUFFER buffer ;
5      D3DXMATERIAL *matl ;
6      //读文件
7      if( D3DXLoadMeshFromX( file , D3DXMESH_SYSTEMMEM ,
8          d3d_Device , NULL ,
9          &buffer , NULL , &m_Num , &m_Mesh ) != D3D_OK )
10     return false ;
11     //配置材质内存
12     m_Texture = new d3dTexture[ m_Num ] ;
13     m_Material = new D3DMATERIAL9[m_Num] ;
14     //取内容
15     matl = (D3DXMATERIAL*)buffer->GetBufferPointer();
16     for( i = 0 ; i < m_Num ; i++ )
17     {
18         m_Material[i] = matl[i].MatD3D ;
19         m_Material[i].Ambient = m_Material[i].Diffuse ;
20         m_Texture[i].Create( matl[i].pTextureFilename );
21     }
22     //法向量
23     if ( !(m_Mesh->GetFVF() & D3DFVF_NORMAL) )
24     {
25         if( m_Mesh->CloneMeshFVF( m_Mesh->GetOptions() &
26             D3DXMESH_32BIT )|D3DXMESH_MANAGED,
27             m_Mesh->GetFVF() | D3DFVF_NORMAL,
28             d3d_Device, &pTempMesh ) != D3D_OK )
29             return 0 ;
30         D3DXComputeNormals( pTempMesh, NULL );
31         m_Mesh->Release();
32         m_Mesh = pTempMesh;
33     }
34     //结束
35     buffer->Release();
36     return true ;
37 }

```

#### 程序说明

(1) 第 7~10 行: 读取 X 模型文件, 返回的信息包含表面颜色信息暂存区 (LPD3DXBUFFER buffer)、模型数量 (m\_Num)、模型顶点信息 (m\_Mesh)。

(2) 第 15~22 行: 将 X 模型文件的材质文件与表面材质信息定义赋给先前声明的全局变量中。

(3) 第 24~34 行: 利用 GetFVF 取得模型顶点样式来判断模型文件内是否有法向量。如果没有, 建立一个内含法向量的新模型, 并把旧模型删除。

(4) 第 36 行: 删除表面颜色信息暂存区。

接下来看图是怎样“画”出来的。

## myd3d.cpp 部分程序代码

```
1 void d3dXFileData::Draw()
2 {
3     DWORD i ;
4
5     for( i = 0 ; i < m_Num ; i++ )
6     {
7         d3d_Device->SetMaterial( &m_Material[i] );
8         d3d_Device->SetTexture( 0 , m_Texture[i] );
9
10        m_Mesh->DrawSubset( i );
11    }
12 }
```

只要把 X 文件中的表面材质与图片材质赋给 Direct Graphics, 再调用 DrawSubset 就可以了。

首先将制作好的对象 d3dXFileData 放在程序中, 在使用之前, 如同拍照一样要先用 Direct Graphics 定义当前环境, 请看以下程序代码。

## canvasFrame.cpp 的部分程序代码

```
1 BOOL canvasFrame::InitWindows()
2 {
3     if( !d3dCreate( m_hWnd , 640 , 480 , true ) )
4         return PostMessage( WM_CLOSE );
5     a_Bk.Create( "背景.tga" );
6     a_Mesh.Create( "tiger.x" );
7     //设定相机位置
8     D3DXVECTOR3 vFrom( 3, 3, -3 );
9     D3DXVECTOR3 vAt( 0, 0, 0 );
10    D3DXVECTOR3 vUp( 0, 1, 0 );
11    D3DXMATRIXA16 matView;
12    D3DXMatrixLookAtLH( &matView, &vFrom, &vAt, &vUp );
13    d3d_Device->SetTransform( D3DTS_VIEW, &matView );
14    //设定投影矩阵
15    D3DXMATRIXA16 matProj;
16    D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4 , 1.0f , 1.0f , 100.0f );
17    d3d_Device->SetTransform( D3DTS_PROJECTION, &matProj );
18    //光
19    d3d_Device->SetRenderState( D3DRS_AMBIENT, 0xFFFFFFFF );
20
21    //
22    return true ;
23 }
```

**程序说明:**

- (1) 第 1 行: 函数在收到消息 WM\_CREATE 时被调用, 这使程序更为简洁。
  - (2) 第 3~6 行: 建立 Direct Graphics 并读入背景图 (背景.tga) 和 X 模型文件 (tiger.x)。
  - (3) 第 8~13 行: 设定视角矩阵 (View), 把视角放在 (3,3,-3) 的位置指向 (0,0,0)。
  - (4) 第 15~17 行: 设定透视投射矩阵 (Projection), 调用 D3DXMatrixPerspectiveFovLH 依序输入 Fov (镜头广角), 厚度、近平面与远平面距离, 再赋给 Direct Graphics。
  - (5) 第 19 行: 设定环境光。
- 最后来看看洗出来的相片效果。

**canvasFrame.cpp 部分程序代码**

```
1 void canvasFrame::Render()  
2 {  
3     //清空  
4     d3dClear();  
5     //开始绘制  
6     d3d_Device->BeginScene();  
7     d3d_Device->SetRenderState( D3DRS_ZENABLE , D3DZB_FALSE );  
8     d3d_Device->SetRenderState( D3DRS_SHADEMODE , D3DSHADE_FLAT );  
9     d3d_Device->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );  
10    a_Bk.BltFast( 0 , 0 );  
11  
12    d3d_Device->SetRenderState( D3DRS_ZENABLE , D3DZB_TRUE );  
13    a_Mesh.Draw();  
14    d3d_Device->EndScene();  
15    //成像  
16    d3d_Device->Present( NULL , NULL , NULL , NULL );  
17  
18 }
```

**运行结果**

程序运行的结果如图 8-4 所示。



图 8-4



## 程序说明

第 13 行就是要画的模型，接着把模型画出来。但是在画之前，因为在画背景时将深度缓冲区（ZBuffer）关闭（第 8 行），所以必须在第 12 行打开它，否则会因顶点无法判别顺序而乱画一通。

## 8.2 3D 空间坐标的转换

在 3D 计算机绘图中，坐标转换是非常重要的课题，因为它决定了对象被如何观察，不同的坐标转换会产生不同的视觉效果，就像是相机放在两个不同的位置去照同一个对象一样，该物体在两张底片上有不同的成像。

要把一个空间中的物体显示到屏幕上，需要掌握坐标转换的动作与过程。本节将探讨 Direct Graphics 的坐标转换（Transform）模块，学习怎样将“未转换”的空间顶点变换成“已转换”的屏幕坐标。

### 8.2.1 Direct Graphics 坐标转换管线

首先设定一个点的坐标值，通常先把模型的某处作为原点，如人体的脚底、桌面的正中心等，其他的顶点按照原点展开，这就是所说的模型坐标系（Model）。

模型坐标系属于几何模型本身，无法描述自己在场景中的位置。在计算机的 3D 环境中，必须根据三种不同作用的 3D 描述效果和一个空间裁割区域，将模型坐标转化成正确的屏幕坐标，分别说明如下：

- （1）3D 世界环境描述（World）
- （2）视角环境描述（View）
- （3）投射环境描述（Projection）
- （4）视端口区域描述（Viewport）

图 8-5 表明了转换的过程。

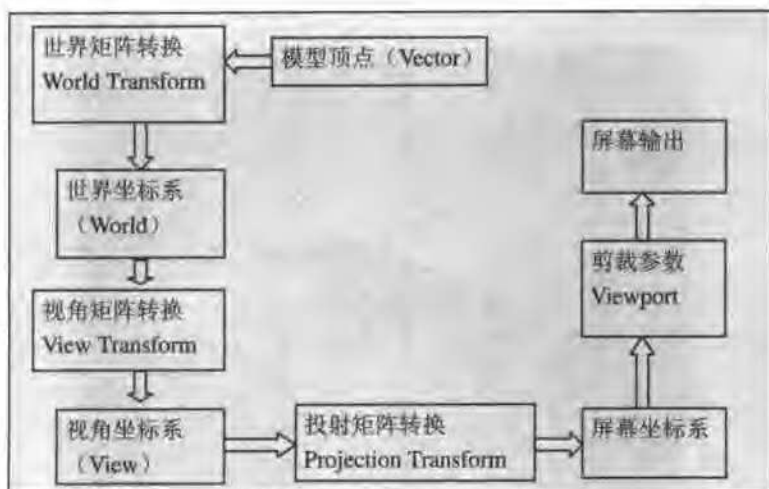


图 8-5

在 Direct Graphics 中, 只要调用 SetTransform 就可以设定空间、视角或投射矩阵。可参看表 8-3。

表 8-3

HRESULT SetTransform( D3DTRANSFORMSTATETYPE State, CONST D3DMATRIX *pMatrix );	
D3DTRANSFORMSTATETYPE State	通知 Direct Graphics 将要设定的矩阵类型, 常用的参数有: D3DTS_WORLD 世界矩阵 D3DTS_VIEW 视角矩阵 D3DTS_PROJECTION 投射矩阵
D3DMATRIX *pMatrix	输入被设定的矩阵, 通常声明为 D3DXMATRIXA16 以方便矩阵计算

## 8.2.2 世界环境描述

一个场景可能会有数百到数千个模型, 也可能要通过缩放、旋转等转换操作, 这时需要另外规定统一的坐标和所有模型共同参考的坐标系统, 使得各对象的位置都能用这种坐标系统来表达, 这就是世界坐标系 (World)。用世界环境定义的矩阵叫做“世界矩阵”。

在 Direct Graphics 中, 可以先对模型进行一连串转换 (例如位移或旋转), 再将所得到的最后矩阵放到 Direct Graphics 的世界矩阵中。可以参看范例 ch8\_2 (见随书光盘) 设定。

### CanvasFrame.cpp

```

1
2 void canvasFrame::Render()
3 {
4     D3DXMATRIXA16 mat, tran, rot;
5     //清空
6     d3dClear();
7     //开始绘制
8     d3d_Device->BeginScene();
9     d3d_Device->SetRenderState( D3DRS_ZENABLE, D3DZB_FALSE );
10    d3d_Device->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_FLAT );
11    d3d_Device->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
12    a_Bk.BltFast( 0, 0 );
13    //开Xbuffer
14    d3d_Device->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
15    //在世界原点画一个对象
16    D3DXMatrixIdentity( &mat );
17    d3d_Device->SetTransform( D3DTS_WORLD, &mat );
18    a_Mesh.Draw();
19    //在世界旋转后位移画一个对象
20    D3DXMatrixRotationY( &rot, D3DX_PI/ 180.0f * 90.0f );
21    D3DXMatrixTranslation( &tran, 1.0f, 0.0f, 1.0f );
22    mat = rot * tran;
23    d3d_Device->SetTransform( D3DTS_WORLD, &mat );
24    a_Mesh.Draw();
25    d3d_Device->EndScene();

```

```
26 //成像
27 d3d_Device->Present( NULL, NULL, NULL, NULL );
28 }
```

## 程序说明

- (1) 第 16~18 行, 将矩阵设为单位矩阵, 传给 Direct Graphics 后画出第一只老虎。
- (2) 第 20~23 行, 先将世界以 Y 为轴旋转 90°, 并且位移到 (x1, z1) 的位置, 两者相乘后, 传给 Direct Graphics 画出第二只老虎。

## 运行结果

程序运行的结果如图 8-6 所示。



图 8-6

有关矩阵内部的运算方法这里再详细说明一下, 可以先使用 Direct Graphics 的工具函数设定矩阵, 再将两个或两个以上的矩阵连续相乘即可。常用的函数如表 8-4 所示。

表 8-4

D3DXMatrixIdentity	将矩阵设为单位矩阵
D3DXMatrixRotationX	以 X、Y、Z 为轴, 输入经度以求旋转矩阵
D3DXMatrixRotationY	(1 经度=PI/180°)
D3DXMatrixRotationZ	(180 经度=PI) (一圈周长=2PI)
D3DXMatrixTranslation	输入偏移量, 求得一个平移矩阵
D3DXMatrixScaling	输入 X、Y、Z 的比例, 求得缩放矩阵

最后, 要注意的是矩阵相乘有一定的先后顺序。范例 ch8\_2 是“先旋转后位移”, 如果顺序颠倒, 产生的结果就会不一样, 例如将

```
mat = rot * tran;
```

改为“先位移后旋转”

```
mat = tran*rot;
```

运行结果就会有完全不同的效果, 如图 8-7 所示。



图 8-7

### 8.3.3 视角环境描述

以观察点为准，Z 轴把观察点所看出去的方向作为正向，使其他顶点位置作为观看点的相对位置，这样屏幕显示才会有参照点，这个点称为观察点（Viewpoint），而这个坐标称为视角坐标系，由视角环境所定义的矩阵叫做“视角矩阵”。如图 8-8 所示。

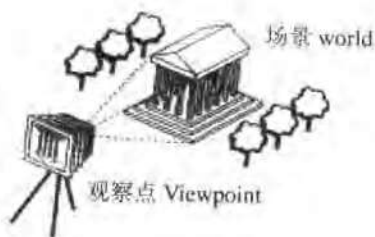


图 8-8

必须先决定视角的位置，再将视角对准特定位置（通常是原点），可以利用 Direct Graphics 的工具函数 `D3DXMatrixLookAtLH`（见表 8-5）求得视角矩阵，再赋给 Direct Graphics。

表 8-5

<code>D3DXMATRIX *D3DXMatrixLookAtLH(</code>	<code>D3DXMATRIX *pOut,</code> <code>CONST D3DXVECTOR3 *pEye,</code> <code>CONST D3DXVECTOR3 *pAt,</code> <code>CONST D3DXVECTOR3 *pUp )</code>
<code>D3DXVECTOR3 *pEye</code>	输出的视角矩阵
<code>D3DXVECTOR3 *pEye</code>	视角的位置
<code>D3DXVECTOR3 *pAt</code>	视角对准的目的点。通常设为原点 (0,0,0)
<code>D3DXVECTOR3 *pUp</code>	视角的向上向量，通常设为 (0,1,0)

## 8.3.4 投射环境描述

一般场景占的范围都相当大，但我们所见的只是一个有限的空间，而投射环境就是把可见场景投影到裁割区域，如同相机的镜头，将前方的景物“拍”到底片中。如图 8-9 所示。

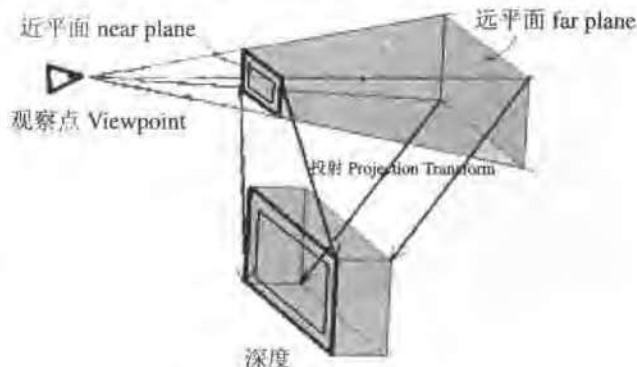


图 8-9

可以调用 `D3DXMatrixPerspectiveFovLH` 函数调整相机的镜头。如表 8-6 所示。

表 8-6

<code>D3DXMATRIX *D3DXMatrixPerspectiveFovLH(</code>	<code>D3DXMATRIX *pOut,</code> <code>FLOAT fovY,</code> <code>FLOAT Aspect,</code> <code>FLOAT zn,</code> <code>FLOAT zf</code> <code>);</code>
<code>D3DXMATRIX *pOut,</code>	转出的投射矩阵
<code>FLOAT fovY</code>	相机广角，通常为近平面与屏幕高度的比例
<code>FLOAT Aspect</code>	深度 (Z)，通常为 1
<code>FLOAT zn</code>	近平面距离
<code>FLOAT zf</code>	远平面距离

### 视端口区域描述 (Viewport)

现在已将场景投影到底片中，最后要做的工作就是将场景“剪”到屏幕上。

简单地说，视图区的意义是在 Direct Graphics 装置描绘完画面之后，呈现在屏幕上的画面可以用在如赛车游戏的后照镜效果或剪成上下两个窗口。

只要设定结构 `D3DVIEWPORT9` 及剪裁区的屏幕位置、长宽及深度，再调用 `SetViewport` 就可以了，如范例 `ch8_3`（见随书光盘）。

**canvasFrame.cpp**

```

1 void canvasFrame::Render()
2 {
3     D3DVIEWPORT9 vp;
```

```

4  //清空
5  d3dClear();
6  //开始绘制
7  d3d_Device->BeginScene();
8      d3d_Device->SetRenderState( D3DRS_SHADEMODE , D3DSHADE_FLAT );
9      d3d_Device->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
10     //屏幕上方第一只老虎
11     vp.X      = 0;
12     vp.Y      = 0;
13     vp.Width  = 640;
14     vp.Height = 240;
15     vp.MinZ   = 0.0f;
16     vp.MaxZ   = 1.0f;
17     d3d_Device->SetViewport( &vp );
18     d3d_Device->SetRenderState( D3DRS_ZENABLE , D3DZB_FALSE );
19     a_Bk.BltFast( 0 , 0 );
20     d3d_Device->SetRenderState( D3DRS_ZENABLE , D3DZB_TRUE );
21     a_Mesh.Draw();
22     //屏幕下方第一只老虎
23     vp.X      = 0;
24     vp.Y      = 240;
25     vp.Width  = 640;
26     vp.Height = 240;
27     vp.MinZ   = 0.0f;
28     vp.MaxZ   = 1.0f;
29     d3d_Device->SetViewport( &vp );
30     d3d_Device->SetRenderState( D3DRS_ZENABLE , D3DZB_FALSE );
31     a_Bk.BltFast( 0 , 0 );
32     D3DXMATRIXA16 mat ;
33     D3DXMatrixRotationY( &mat , D3DX_PI );
34     d3d_Device->SetTransform( D3DTS_WORLD , &mat );
35     d3d_Device->SetRenderState( D3DRS_ZENABLE , D3DZB_TRUE );
36     a_Mesh.Draw();
37 d3d_Device->EndScene();
38 //成像
39 d3d_Device->Present( NULL , NULL , NULL , NULL );
40 }

```

#### 程序说明

(1) 第 11~17 行, 将剪裁空间定在上半部, 画出上半背景及老虎。

(2) 第 23~32 行, 将剪裁空间定在下半部, 并将老虎旋转 180° (经度为  $\pi$ ), 画出背景及老虎。

#### 运行结果

程序运行的结果如图 8-10 所示。

有关 Direct Graphics 的转换模块就介绍到这里, 这里只用到 D3DX 的矩阵工具函数, 并没有使用任何数学运算。虽然它可以帮我们完成非常多的工作, 但希望读者不妨找本相关的书籍, 继续深入地研究矩阵的内部运算。



图 8-10

## 8.4 Direct Graphics 的色彩计算

在计算机图形学中，色彩由光的三原色（RGB）构成，而一个模型的色彩就没有这么简单了，一般希望能通过光的照射来达到更自然的色彩及更真实的场景。先来看看计算机如何根据法向量、光与表面材质达到类似真实世界所呈现的色彩效果。

### 8.4.1 颜色的决定因素

一个几何模型的颜色，主要是由光源、表面材质和覆盖在上面的贴图三者共同决定。

可以设定各种发射性光源及环境光源，各个发射光基本上都要设定位置及颜色，有些还要设定照射方向，而环境光在整个系统中只有一个，这些设定会影响到场景里的所有模型。

在 Direct Graphics 里，光源分为以下几种类型。

- （1）发射光（Emitted Light），由一种对象产生，该对象主动将光粒子发射出去，如电灯泡。
- （2）环境光（Ambient Light），发射光经过一连串物体，由空气中的微尘彼此散射与反射，形成没有方向性的光，并均匀地散布在整个空间中。
- （3）散射光（Diffuse Light），通过光的照射，由各种方向均匀散布出的光线，一般物体所看到的颜色，通常是散射光。
- （4）反射光（Specular Light），当光线照到该物品时，会在相对于法向量的地方反射出去，如光线照到金属或镜子时，在特定的角度会看到光亮的区域。反射性越好，亮区就越小，就越接近白色。

### 8.4.2 发射光的设定方式

Direct Graphics 会帮助计算颜色并产生扩散色（diffuse）与反射色（specular），只要设定好光与材质就可以了，参见范例 ch8\_4（见随书光盘）。

## canvasFrame.cpp

```
1 void canvasFrame::SetLight()  
2 {  
3     //环境光  
4     d3d_Device->SetRenderState( D3DRS_AMBIENT, D3DCOLOR_XRGB( 64 , 64 , 64 ));  
5     //设置点光源  
6     D3DLIGHT9 light ;  
7     memset( &light , 0 , sizeof( light ));  
8     light.Type = D3DLIGHT_POINT ;  
9     light.Diffuse.r = 1 ;  
10    light.Diffuse.g = 1 ;  
11    light.Diffuse.b = 1 ;  
12    light.Position.x = 10 ;  
13    light.Position.y = 10 ;  
14    light.Position.z = 10 ;  
15    D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction,  
16                      (D3DXVECTOR3*)&light.Position );  
17    light.Range = 20 ;  
18    light.Attenuation1 = 0.01f ;  
19    d3d_Device->SetLight( 0 , &light );  
20    d3d_Device->SetRenderState( D3DRS_LIGHTING, TRUE );  
21    d3d_Device->LightEnable( 0 , true );  
22 }
```

## 程序说明

- (1) 第 4 行, 设置环境光。
- (2) 第 6~19 行, 设置一个位置在 (10,10,10) 距离为 20, 光的衰减情况为 0.01 的点光源。
- (3) 第 20 行, 打开光源计算。
- (4) 第 21 行, 打开第 0 组发射光光源计算。

## 运行结果

程序运行的结果如图 8-11 所示。



图 8-11

模型的块状很明显, 那是因为使用的涂色模式为平面 (Flat) 计算, 只要在绘制时将



```
d3d_Device->SetRenderState( D3DRS_SHADEMODE , D3DSHADE_FLAT );
```

改为内差计算 (GOURAUD) 就可以了, 代码为:

```
d3d_Device->SetRenderState( D3DRS_SHADEMODE , D3DSHADE_GOURAUD );
```

## 运行结果

程序运行的结果如图 8-12 所示。



图 8-12

平面 (Flat Shading) 计算是把第一个顶点的颜色当做整个三角形上所有的颜色, 这种涂色模式最快, 原理也最简单, 但失真度较高。

Gouraud 氏计算 (Gouraud Shading) 是由三个顶点的颜色以内插法算出来的。面上的点与某顶点越近的, 则越接近该顶点的颜色, 速度较平面计算稍微慢, 但速度与表现效果是可以接受的。

除了平面与 Gouraud 氏计算外, 还可以使用 Phong 氏计算 (Phong Shading), 这种涂色法是按照材质、法向量及光源的关系计算面上所有点的颜色, 这种涂色模式效果非常逼真, 但非常慢, 如果硬件不支持 Phong 氏计算可能会相当吃力。

结构 D3DLIGHT9 的说明见表 8-7。

表 8-7

D3DLIGHT9	
D3DLIGHTTYPE Type	光源样式, 可以设定为 D3DLIGHT_POINT (点光源) D3DLIGHT_SPOT (聚光灯) D3DLIGHT_DIRECTIONAL (方向光)
D3DCOLORVALUE Diffuse	扩散色的饱和度
D3DCOLORVALUE Specular	反射色的饱和度
D3DCOLORVALUE Ambient	环境色的饱和度
D3DVECTOR Position	光源位置
D3DVECTOR Direction	光的方向
float Range	可达到的最远距离
float Falloff	聚光灯内部圆锥 (umbra) 至外部圆锥 (penumbra) 的衰减情况

(续表)

Float Attenuation0; Float Attenuation1; Float Attenuation2;	光的衰减因子
Float Theta;	聚光灯内部圆锥的角度
Float Phi;	聚光灯外部圆锥的角度

### 8.4.3 表面材质的设定方法

表面材质 (Material), 就是物体表面对照射光的反应。在 Direct Graphics 之中, 材质是一种绘图状态的信息, 可以增强金属光泽或是塑料质感。

在 Direct Graphics 中, 可以设定的参数见表 8-8。

表 8-8

D3DMATERIAL9	
D3DCOLORVALUE Diffuse	对于光源的扩散色的反射情形
D3DCOLORVALUE Ambient	对环境光的反应, 通常设为 1 表示全部反射
D3DCOLORVALUE Specular	对于光源的反射色的反射情形
D3DCOLORVALUE Emissive	材质本身所发出的颜色, 设定此参数看起来就好像变成了一个发射光源, 但它不会影响其他对象的照射情形
Float Power	对反射光的效应强度, 数值越大, 越像金属; 数值越小, 越像塑料

下面试着制作一只“金属虎”, 首先在设定光源的函数 IDirect3DDevice9::SetLight 里, 加入红色的反射光, 打开范例 ch8\_5 (见随书光盘) 的 canvasFrame.cpp, 添加

```
light.Specular.r = 1 ;
light.Specular.g = 1 ;
light.Specular.b = 1 ;
```

在绘制函数里, 打开反射光计算

```
d3d_Device->SetRenderState( D3DRS_SPECULARENABLE , true );
```

#### 运行结果

程序的运行结果如图 8-13 所示。



图 8-13

再根据材质使它更像金属，参考范例 ch8\_6（见随书光盘）的 myD3D.cpp，在加载模型的函数里调整模型表面材质的反射强度。

```
m_Material[i].Power = 20 ;
```

## 运行结果

程序的运行结果如图 8-14 所示。



图 8-14

也可以加入基台操作形成室内窗户对阳光照射的效果。

```
d3d_Device->SetTextureStageState( 0 , D3DTSS_COLORARG1 , D3DTA_TEXTURE );  
d3d_Device->SetTextureStageState( 0 , D3DTSS_COLORARG2 , D3DTA_SPECULAR );
```

## 运行结果

程序的运行结果如图 8-15 所示。



图 8-15

由于反射光的计算涉及的数学原理相当复杂，非常消耗时间，所以在不计算反射光的情况下，应调用

```
d3d_Device->SetRenderState( D3DRS_SPECULARENABLE , false );
```

来关闭它。

第 7 章和本章中已经介绍完 Direct Graphics 的基本绘图原理与作法。本书着重于“演练”，故省略了许多和绘图相关的公式。当然，3D 的领域不是短短的两章内容可讲完的，如果读者有兴趣的话，可以参阅一些 3D 绘图的书，研究更高的 3D 领域。

## 课后重点整理

- 3D 是指被描述或显示的对象有宽度、高度与深度三个测量维度，又称为三维。
- Direct Graphics 的三大模块，根据坐标转换（Transform）画出苹果的“型”，根据色彩计算（Lighting）决定苹果的颜色，根据平面绘制（Raster）将照片洗出来。
- 3D 绘图中的模型是指用数据表示顶点的集合，可以是有具体形态的物体（如书本、计算机）或无具体形态的动态（如浪花、光线）等，这些顶点都可以存于文件中。
- 构成一个 3D 虚拟环境的最基本原则是，它必须有一个 3D 空间的环境坐标系统。这种环境坐标系统分为两大类：右手 3D 坐标系统和左手 3D 坐标系统。
- 要用 Direct Graphics 计算光，必须要提供以下信息。
  - （1）顶点法向量（Normal）。
  - （2）环境光或发射光（Light）。
  - （3）模型表面材质（Material）。
- 一个几何模型的颜色，主要是由光源、表面材质和覆盖在上面的贴图三者共同决定。

### 课后练习

1. 模型坐标系属于几何模型本身，而无法用于描述自己在场景中的位置。在计算机的 3D 环境里，必须通过三种不同作用的 3D 描述效果及一个空间裁割区域，才能将模型坐标转成正确的屏幕坐标，请简单说明之。
2. 什么是世界坐标系（World）？什么是视角坐标系？
3. 在 Direct Graphics 里，光源可分为哪几种类型？
4. 试比较平面计算（Flat Shading）、Gouraud 氏计算（Gouraud Shading）、以及 Phong 氏计算（Phong Shading）三种涂色模式的主要原理。



## 第9章 DirectSound 的使用方式

在 Windows 多媒体发展初期，提供 Mci 的套件。通过 Mci 套件中如 mciSendString 等函数，软件工程师能直接输入简单的文字，“命令” Windows 与设备无关的特性来播放多媒体。

随着时代的进步，视觉艺术从早期的简单贴图发展到现在的混色和 3D 技术，声音也从早期的单音道，到现在的混音与环绕音效。Mci 就如同 DGI，虽然简单，但满足不了玩家的胃口。

在学会了 Direct Graphics 之后，相信各位已经深深地感受到 DirectX 的威力。在这一章里要来介绍的是 DirectX 中的另一成员“DirectSound”。

### 9.1 开始建立 DirectSound

DirectSound 提供了各种音效处理的支持，如低延迟音、3D 立体音、协调硬件运作等音效功能。

在这一章中先介绍如何利用 DirectSound 在程序中加入声音，首先来了解 DirectSound 程序的建立流程与步骤。

#### 9.1.1 建立 DirectSound 的第一步

建立 DirectSound 的第一个步骤是确认在 VC++ 中是否已经设定好引用 DirectX 头文件与函数库的所在路径。

接下来设定 DirectSound 程序连接时必须用到的函数库与头文件，如图 9-1 和图 9-2 所示。

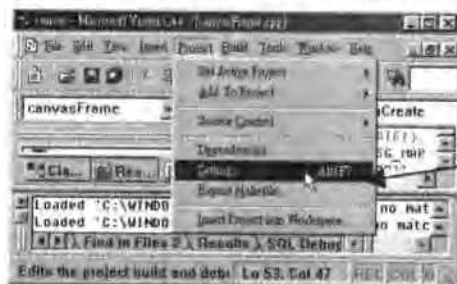


图 9-1

输入编辑时要连接的函数库为“Dxguid.lib”、“Dsound.lib”与“Winmm.lib”。

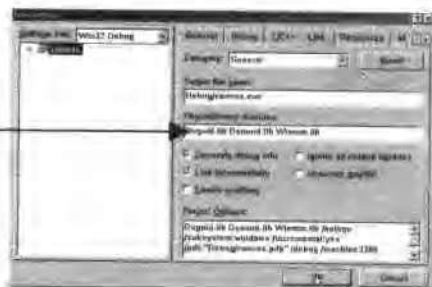
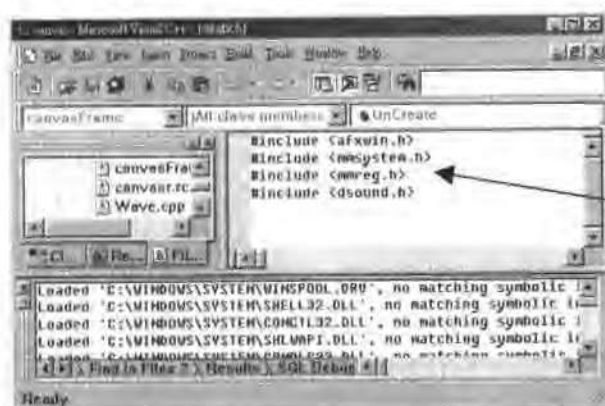


图 9-2

除了建立标准的 DirectSound 对象所需的“Dxguid.lib”与“Dsound.lib”函数库之外，其中“Winmm.lib”是 Windows 提供的函数库，用来处理多媒体文件的函数库，由于加载 WAVE 文件时会用到，所以在这里必须将其加入项目的连接中。接下来打开项目中的“stdafx.h”，加入要引用的头文件，如图 9-3 所示。



输入头文件“mmsystem.h”、“mmreg.h”及“dsound.h”

图 9-3

在此必须引用“mmsystem.h”、“mmreg.h”、“dsound.h”三个头文件，这些头文件是 DirectSound 程序要用到的。在完成了以上这些项目的各项设定之后，接下来就可以开始试着建立与使用 DirectSound 程序了。

### 9.1.2 DirectSound 对象的建立

建立 DirectSound 系统必须经过下列步骤。

- (1) 建立 DirectSound 对象。
- (2) 设定共享层级。
- (3) 设定主缓冲区的格式。

首先要建立一个代表声卡的 DirectSound 对象。在 OnCreate 函数中加入以下的程序代码来建立 DirectSound 对象，在范例 ch9\_1 中（见随书光盘）可看到以下程序代码：

#### 程序代码

```
1  LPDIRECTSOUND pDS; //声明 DirectSound 对象指针
2  HRESULT result; //声明 HRESULT 类型变量
3  result = DirectSoundCreate( NULL, &pDS, NULL ); //建立 DirectSound 对象
4  if(result != DS_OK)
5  MessageBox("建立 DirectSound 对象失败!");
```

#### 程序说明

- (1) 第 1 行：声明一个 LPDIRECTSOUND 类型的 IDirectSound::DirectSound 对象指针“pDS”。
- (2) 第 2 行：声明一个 HRESULT 类型的变量，存储运行 DirectX 方法后的返回值，判断运行是否成功。
- (3) 第 3 行：程序代码调用“DirectSoundCreate”建立 DirectSound 对象，参数 1 设为“NULL”，表示使用目前预设的声卡，也可以调用 DirectSoundEnumerate 取得可用的声卡；参数 2 输入对象的指针；参数 3 必须设为“NULL”，运行此方法后返回值赋给 result。

(4) 第4行: 判断返回值 result 是否等于“DS\_OK”, 若是, 则表示建立对象成功, 否则显示错误消息。

DirectSound 会自动判断声卡硬件的配备, 如果有硬件加速功能则会自动运用, 并且尽可能地使用声卡上的内存进行各种声音效果的处理。

### 9.1.3 设定程序协调层级

DirectSound 使用到声卡, 而且声卡也是 Windows 控制的资源之一, 其他程序随时都可以使用它。因此必须在建立了 DirectSound 对象后, 顺便设定协调层级来告诉 Windows 使用硬件的权限, 其他应用软件与游戏如何共享。下面的程序代码便是设定 DirectSound 对象的协调层级:

#### 程序代码

```
1 result = pDS->SetCooperativeLevel( m_hWnd, DSSCL_PRIORITY );
2 if(result != DS_OK)
3     MessageBox("设定程序协调层级失败!");
```

在上面的程序代码中, 调用“SetCooperativeLevel”设定协调层级, 其中参数1设为“m\_hWnd”, 表示为应用程序主窗口; 参数2为设定标志位(flag), 设定程序使用资源的优先权, 可选标志位如表9-1所示。

表 9-1

设定标志位	说 明	缺 点
DSSCL_EXCLUSIVE	独占模式, 当调用程序时, 只有此程序可使用播放声音的资源	背景应用程序都处于静音状态
DSSCL_NORMAL	正常模式, 播放声音的资源可与其他程序共享	无法改变 DirectSound 主要格式
DSSCL_PRIORITY	可设定主缓冲区的播放模式	可能改变其他软件的输出格式
DSSCL_WRITEPRIMARY	具有最高优先权, 可直接存取主缓冲区的内容	辅助缓冲区将无法播放, 其他应用软件会失去所属的暂存区

### 9.1.4 缓冲区的基本概念

Direct Graphics 使用“影像材质”(Texture)显示图片, 而在 DirectSound 中则是使用“声音缓冲区”播放声音, 影像材质与声音缓冲区在 Direct Graphics 与 DirectSound 中所扮演的角色其实有点相似。这一小节将介绍 DirectSound 中使用的两种缓冲区类型。

#### 1. 主缓冲区 (Primary Buffer)

“主缓冲区”可以看做一个 DirectSound, 是用来播放声音、产生混音效果的区域, 它有一个预设的播放格式(8-bit、22 kHz), 而声音文件在播放时便按照这种格式做输出。

主缓冲区在建立 DirectSound 对象时自动生成, 因此不需要专门建立主缓冲区。不过若需要比默认值更好地播放品质, 如 16-bit、44kHz, 就必须建立主缓冲区并设定其播放的格式, 并且在前面设定协调层级时, 标志位必须设定为“DSSCL\_PRIORITY”。

对于设定的播放格式, 若使用者的声卡不支持, 并不会影响程序的运行。DirectSound 会自动按照使用者的计算机配置来调整最佳的输出音效, 这个优点也省去了程序设计者检测各不同使用者、



不同计算机硬件配备的麻烦。

## 2. 次缓冲区 (Secondary Buffer)

“次缓冲区”存储播放声音的文件，可以建立数个次缓冲区来存放多个要播放的声音文件。

当建立了次缓冲区并在其中加载声音文件之后，便可播放其中的内容，而播放的方式很简单，只要调用次缓冲区对象的“Play”方法，声音就会自动地送入主缓冲区中并进行播放，下面以图 9-4 进行说明。



图 9-4

如果在同一时间内播放多个声音文件（例如同时播放“背景音乐”与“事件声音”），那么播放的方法也相同。各个存储声音的次缓冲区调用“Play”方法，声音就会送入主缓冲区中，接着在主缓冲区中对这些声音进行“混音”操作，最后再输出。如此就可于同一时间听到多种不同声音的播放，下面看看图 9-5 所示的说明。



图 9-5

次缓冲区中所存储的声音不仅仅可以进行播放，还可以直接调用方法来进行改变音量、调整声道、停止或循环播放等操作。

### 9.1.5 建立主缓冲区

清楚了 DirectSound 中缓冲区的概念之后，下面就来说明建立主缓冲区与设定播放格式的方法，这个步骤并非绝对必要。

#### 1. 建立主缓冲区

首先来看看建立主缓冲区的程序代码：

##### 程序代码

```
1 LPDIRECTSOUNDBUFFER pPBuf;           //声明主缓冲区指针
2 DSBUFFERDESC desc;                   //声明描述结构
```

```

3  memset( &desc,0, sizeof(desc) );           //清空结构内容
4  desc.dwSize = sizeof(desc);                 //配置描述结构大小
5  desc.dwFlags = DSBCAPS_PRIMARYBUFFER;
6  desc.dwBufferBytes = 0;
7  desc.lpwfxFormat = NULL;
8  result = pDS->CreateSoundBuffer( &desc, &pPBuf, NULL );
9  if(result != DS_OK)
10     MessageBox("建立主缓冲区失败!");

```

**程序说明**

(1) 第 1 行: 声明一个缓冲区的对象指针“pPBuf”。

(2) 第 2 行: 定义一个“DSBUFFERDESC”类型的结构“desc”，用于设定缓冲区的各项特性。

(3) 第 5 行: 设定 desc 的“dwFlags”资料成员为“DSBCAPS\_PRIMARYBUFFER”，表示为主缓冲区。

(4) 第 6、7 行: 设定资料成员“dwBufferBytes (缓冲区大小)”、“lpwfxFormat (缓冲区格式)”，主缓冲区的这两项设定必须为“0”及“NULL”。

(5) 第 9 行: 调用“CreateSoundBuffer”方法并依次输入三个参数“desc (缓冲区特性)”、“pPBuf (缓冲区对象指针)”、“NULL (一定要设为 NULL)”，建立主缓冲区。

**2. 设定播放格式**

通过一段的程序代码建立了一个主缓冲区，而在前面通过设定主缓冲区的播放格式改变声音播放的品质与效果。设定播放格式必须使用“SetFormat”方法，下面是设定主缓冲区播放格式的程序内容：

**程序代码**

```

1  WAVEFORMATEX pwfmt;                         //声明声音结构
2  memset( &pwfmt,0, sizeof(pwfmt) );
3  pwfmt.wFormatTag = WAVE_FORMAT_PCM;
4  pwfmt.nChannels = 2;                        //播放声道
5  pwfmt.nSamplesPerSec = 44100;               //播放频率
6  pwfmt.wBitsPerSample = 16;                  //位
7  pwfmt.nBlockAlign = pwfmt.wBitsPerSample / 8 * pwfmt.nChannels;
8  pwfmt.nAvgBytesPerSec = pwfmt.nSamplesPerSec * pwfmt.nBlockAlign;
9  result = pPBuf->SetFormat(&pwfmt);          //设定播放格式
10 if(result != DS_OK)
11     MessageBox("设定播放格式失败!");

```

**程序说明**

(1) 第 1 行: 定义了一个“WAVEFORMATEX”类型的结构“pwfmt”，描述播放 WAVE 档的各项特性。

(2) 第 3~8 行: 设定 pwfmt 的内容，在这里设定要播放的声音类型为“2 声道”、“PCM”、“44.1kHz”、“16bit”。

(3) 第 9 行: 调用“SetFormat”方法并输入“pwfmt”，设定主缓冲区的播放格式。

完成了前面主缓冲区的建立与设定的步骤，当播放声音时，DirectSound 便会自动将声音以设定的播放格式输出。

## 9.1.6 WAVE 声音文件的加载

学会建立主缓冲区后，接下来开始建立次缓冲区，在 DirectSound 程序中至少有一个次缓冲区存储要播放的声音。而次缓冲区的建立必须要参考加载的声音（.wav）文件的两种信息：“声音文件格式”和“波形资料大小”。

因此在建立次缓冲区之前，必须先加载一个声音文件并取得设定次缓冲区参考的信息及声音的波形资料。

这一小节里介绍如何在 VC++ 中读入一个 WAVE 声音文件，取得其中的信息与播放的资料。

### 1. WAVE 声音文件的格式

WAVE 声音文件是符合 RIFF（Resource Interchange File Format）规格的一种多媒体文件，RIFF 规格的文件是利用“区块（chunk）”方式存储文件的，包含记录文件格式的“格式区块（fmt）”与文件实际内容的“资料区块（data）”，而这两个区块又都是 RIFF 区块的子区块。下面以图 9-6 所示来进行说明。

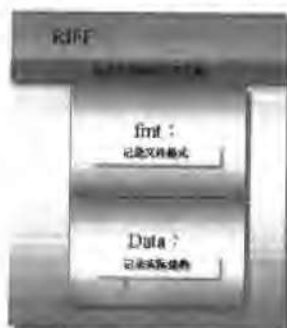


图 9-6

其中代表各个区块的名称字符串都可在文件中找到，如此便得以辨认区块及取得该区块中的资料。如图 9-7 所示是以二进制格式打开一个声音文件显示的内容。



图 9-7

### 2. 读取文件内容

看过 WAVE 声音文件的格式之后，接下来说明应该如何读取声音文件的内容。

读取多媒体文件会用到 Windows 所提供的多媒体函数库“Winmm.lib”并引用必要的头文件，这些在 9.1.1 小节中已说明，下面直接看看在 VC++ 中打开与加载 WAVE 文件的流程与步骤。

- (1) 打开文件。
- (2) 确认是否为 RIFF 文件，类型为 WAVE。
- (3) 寻找 fmt 区块，取得文件格式。
- (4) 寻找 data 区块，取得文件内容。
- (5) 关闭文件。

以下就是按照上面的流程步骤加载 WAVE 声音文件资料的程序内容，先来看变量声明的部分。

#### 结构

```

1  WAVEFORMATEX  swfmt;           //声明声音结构
2  MMCKINFO      ckRiff;          //RIFF 区块的信息
3  MMCKINFO      ckInfo;          //子区块的信息
4  MMRESULT      mmresult;        //返回的结果
5  HMMIO         hmmio;           //打开的多媒体文件
6  DWORD         size;            //实际资料大小

```

#### 结构说明

(1) 第 1 行：设定一个“WAVEFORMATEX”类型的结构“swfmt”，存储读出的 WAVE 声音文件格式。

(2) 第 2、3 行：设定“MMCKINFO”类型的结构“ckRiff”与“ckInfo”，分别代表母区块 (RIFF) 与子区块 (fmt 或 data) 的信息。

(3) 第 4 行：设定“MMRESULT”类型的结构“mmresult”，存储运行函数后返回的结果。

(4) 第 5 行：设定“HMMIO”类型的结构“hmmio”，代表所打开的多媒体文件。

接下来就利用上述的结构判断打开的文件是否为 WAVE 类型 (RIFF 区块中必须有 WAVE 字符串)，并搜寻其中是否有 fmt 和 data 区块。若是格式正确的 WAVE 文件，则取出设定次缓冲区时所需的“文件格式”和“资料大小”，分别记录在 swfmt 与 size 中，下面是程序的详细说明。

#### 程序代码

```

1  hmmio = mmioOpen("sound.wav", NULL,
2              MMIO_ALLOCBUF | MMIO_READ );           //打开文件
3  if(hmmio == NULL)                                   //判断是否为空
4      MessageBox("文件不存在!");
5  ckRiff.fccType = mmioFOURCC('W', 'A', 'V', 'E');
6  //设定文件类型
7  mmresult = mmioDescend(hmmio, &ckRiff, NULL, MMIO_FINDRIFF);
8  //搜寻类型
9  if(mmresult != MMSYSERR_NOERROR)
10     MessageBox("文件格式错误!");
11  ckInfo.ckid = mmioFOURCC('f', 'm', 't', ' ');      //设定区块类型
12     mmresult = mmioDescend(hmmio, &ckInfo, &ckRiff, MMIO_FINDCHUNK);
13 //搜寻区块
14 if(mmresult != MMSYSERR_NOERROR)
15     MessageBox("文件格式错误!");
16 if(mmioRead(hmmio, (HPSTR)&swfmt, sizeof(swfmt)) == -1)
17     MessageBox("读取格式失败!");

```

```

18 mmresult = mmioAscend(hmmio,&ckInfo,0);           //跳出子区块
19 ckInfo.ckid = mmioFOURCC('d','a','t','a');        //设定区块类型
20 mmresult = mmioDescend(hmmio,&ckInfo,&ckRiff,MMIO_FINDCHUNK);
21 //搜寻区块
22 if(mmresult != MMSYSERR_NOERROR)
23     MessageBox("文件格式错误!");
24 size = ckInfo.cksize;                             //取得实际资料大小
25 mmioClose(hmmio,0);

```

## 程序说明

- (1) 第 1、2 行：使用“mmioOpen”函数打开“sound.wav”文件，并赋给 hmmio。
- (2) 第 5 行：设定“ckRiff”的“fccType”成员为“WAVE”，即 RIFF 文件的类型必须为 WAVE。
- (3) 第 7 行：使用“mmioDescend”搜寻区块下的资料，并把返回结果赋给 mmresult，其中参数设定说明见表 9-2。

表 9-2

参 数	说 明
hmmio	打开的文件
&ckRiff	RIFF 区块信息，fccType 成员设为 WAVE
NULL	区块的母区块，此处为设为 NULL，因为 RIFF 为最上层的区块
MMIO_FINDRIFF	设定此参数表示在 RIFF 区块中搜寻

该行程序代码在 RIFF 区块中搜寻是否含有字符串 WAVE，若找到该字符串则表示为 WAVE 类型文件。若找不到，mmioDescend 函数返回“MMSYSERR\_NOERROR”。

- (4) 第 11 行：若找到了 RIFF 区块下的 WAVE 字符串，程序代码定义另一个描述区块数据结构“ckInfo”的“ckid”成员为“fmt”，第 4 个字符为空格，即一个名称为“fmt”的区块。

- (5) 第 12 行：同样使用 mmioDescend 函数在 RIFF 中搜寻 fmt 区块，参数设定说明见表 9-3。

表 9-3

参 数	说 明
hmmio	打开的文件
&ckInfo	搜寻区块的信息，名称设为 fmt
&ckRiff	搜寻的母区块信息，输入&ckRiff 表示母区块为 RIFF
MMIO_FINDCHUNK	设定此参数表示搜寻了区块

- (6) 第 16 行：若找到了 fmt 区块，程序代码会使用“mmioRead”函数读出其内容，存入 swfmt。
- (7) 第 18 行：跳出目前所在的 fmt 子区块。
- (8) 第 19、20 行：按照相同的方法在 RIFF 区块中搜寻“data”子区块，若找到所指定的 data 区块，就把该区块的相关信息记录在 ckInfo 中。
- (9) 第 24 行：取得 ckInfo.cksize 的值，设定给变量 size，即声音文件资料的实际大小。
- (10) 第 25 行：使用完打开的多媒体文件之后，将其关闭。

以上的程序代码取得两项设定次缓冲区时必要的资料 swfmt（格式）与 size（大小），而实际编写程序时，第 24、25 行之间还要加入一些建立次缓冲区的程序代码，并且读取 data 区块中要播放的声音文件的资料。

### 9.1.7 建立次缓冲区

在取得声音文件的格式与资料大小后，接着就可以建立存储声音的次缓冲区了，建立次缓冲区的程序代码如下：

#### 程序代码

```

1  LPDIRECTSOUNDBUFFER pSBuf;           //声明次缓冲区指针
2  memset( &desc,0,sizeof(desc));       //清空结构内容
3  desc.dwSize = sizeof(desc);           //配置结构大小
4  desc.dwBufferBytes = size;             //设定缓冲区大小
5  desc.lpwfxFormat = &swfmt;            //设定缓冲区格式
6  desc.dwFlags = DSBCAPS_STATIC | DSBCAPS_CTRLPAN |
7  DSBCAPS_CTRLVOLUME| DSBCAPS_GLOBALFOCUS;
8  result = pDS->CreateSoundBuffer( &desc, &pSBuf, NULL );
9  if(result != DS_OK)
10     MessageBox("建立次缓冲区失败!");

```

#### 程序说明

- (1) 第 2、3 行：清空并重设 desc 结构的大小，定义次缓冲区的特性。
- (2) 第 4、5 行：利用前面所取得的声音文件格式（swfmt）与大小（size），设定次缓冲区的格式与大小。
- (3) 第 6、7 行：设定次缓冲区的特性，表 9-4 列出了可用的设定标识位及其说明。

表 9-4

设定标志位	说 明
DSBCAPS_CTRL3D	具有 3D 音效效果
DSBCAPS_CTRLFREQUENCY	可控制频率
DSBCAPS_CTRLPAN	可控制声道
DSBCAPS_CTRLVOLUME	可控制音量
DSBCAPS_GLOBALFOCUS	设定此标识位，当程序切换至其他程序（使用 DirectSound 的程序）时，仍可继续播放声音
DSBCAPS_LOX'DEFER	可使用硬件或者软件来播放声音，若要使用 Direct 7.0 新的声音定位与管理功能，必须设定此标志位
DSBCAPS_LOCHARDWARE	强迫使用硬件内存，若硬件不支持则建立缓冲区失败
DSBCAPS_LOCSOFTWARE	强迫使用软件内存，并由软件来进行混音
DSBCAPS_MUTE3DATMAXDISTANCE	超过最大可听距离，自动停止播放
DSBCAPS_PRIMARYBUFFER	主缓冲区
DSBCAPS_STATIC	可重复播放，并自动使用可用硬件资源
DSBCAPS_STICKYFOCUS	设定此标识位，当程序切换至其他程序（不使用 DirectSound 的程序）时，仍可继续播放声音

- (4) 第 8 行：调用“CreateSoundBuffer”方法来建立次缓冲区“pSBuf”。

在建立了次缓冲区之后，便可以将要播放的音文件资料读入缓冲区中，进行声音的播放与音效的处理。

## 9.1.8 加载声音到次缓冲区

要将声音加载到次缓冲区中，等于对缓冲区进行写入的动作，因此必须先锁定缓冲区，等待完成加载的操作后再解除锁定。下面是在次缓冲区中加载声音资料的程序代码。

### 程序代码

```

1  LPVOID pAudio;
2  DWORD bytesAudio;
3  result = pSBuf->Lock(0,size,&pAudio,&bytesAudio,NULL,NULL,NULL);
4  //锁定缓冲区
5  if(result != DS_OK)
6  MessageBox("锁定缓冲区失败!");
7  mmresult = mmioRead(hmmio, (HPSTR)pAudio,bytesAudio);
8  //读取声音文件资料
9  if(mmresult == -1)
10     MessageBox("读取声音文件资料失败!");
11 result = pSBuf->Unlock(pAudio,bytesAudio,NULL,NULL);
12 //解除锁定缓冲区
13 if(result != DS_OK)
14 MessageBox("解除锁定缓冲区失败!");
15 mmioClose(hmmio,0);

```

### 程序说明

- (1) 第 1 行：声明一个指针“pAudio”，指向缓冲区中要开始记录资料的地址。
- (2) 第 2 行：声明一个变量“bytesAudio”，存储缓冲区中要存入资料的长度。
- (3) 第 3 行：调用“Lock”方法锁定次缓冲区，其中各个参数设定的意义说明见表 9-5。

表 9-5

参 数	说 明
0	要锁定缓冲区的开始位置，0 代表从最前面开始
size	要锁定缓冲区的大小
&pAudio	指向缓冲区的指针
&bytesAudio	缓冲区中要存入资料的长度
NULL	第一个指向缓冲区的指针，设为 NULL 表示不划分第二个区块
NULL	缓冲区中第二个区块要存入资料的长度
NULL	设定标识位，此处设为 NULL

(4) 第 7 行：运行“mmioRead”函数，输入指针（pAudio），读取资料的长度（bytesAudio），读取打开的多媒体文件 hmmio 中 data 区块的资料。这样就在缓冲区中加载了要播放的声音资料。

(5) 第 11 行：调用“Unlock”方法解除锁定缓冲区。

(6) 第 15 行：关闭打开的文件。

通过本节中介绍的流程与步骤，完成了 DirectSound 程序的基本架构，而且也在缓冲区中加载了要使用的声音文件资料。

## 9.2 声音的播放与控制

如果已经成功地建立次缓冲区并加载声音文件资料,那么接下来有关声音的播放与控制工作可以说是轻松简单了。

### 9.2.1 播放声音功能

这一小节里将介绍“Play”与“Stop”两个方法,它们分别是次缓冲区用来播放与停止声音的方法。下面直接用程序代码说明使用的方法。

```
pSBuf->Play(0,0,1);           //循环播放声音
pSBuf->Stop();                 //停止播放声音
```

在上述的程序代码中,Play方法的第3个参数设为1,表示循环播放缓冲区“pSBuf”中的声音,若设为0,则仅播放一次声音;Stop方法是停止pSBuf缓冲区的声音播放。

### 9.2.2 制作混音功能

混音就是将多种声音的波形组合形成一个新的波形。以两种不同声音的波形为例,混音是将两种波形组合为一,形成另一种新波形,播放新波形的资料时会听到原先两种声音一起播放的效果。

在DirectSound中可以同时播放数个声音,混音的操作是由DirectSound将各个次缓冲区的声音资料复制到主缓冲区中并自动进行混音的工作,然后再将其输出,让人感觉到多组声音一起播放的效果。参见范例ch9\_1(见随书光盘)。

在这个范例中,将DirectSound初始化的程序代码写在OnCreate函数中,同时编写一个“createbuffer”函数为每一个打开的声音文件建立一个次缓冲区,并将声音文件加载以供程序使用,下面看看程序的内容。

#### 程序代码

```
1  LPDIRECTSOUNDBUFFER canvasFrame::createbuffer(char* filename)
2  {
3      LPDIRECTSOUNDBUFFER buf;
4      hmmio = mmioOpen(filename, NULL, MMIO_ALLOCBUF
5      |MMIO_READ );
6      if(hmmio == NULL)
7          MessageBox("文件不存在!");
8      //判断与取得WAVE文件格式的程序代码
9      memset( &dsdesc,0,sizeof(dsdesc));
10     dsdesc.dwSize = sizeof(dsdesc);
11     dsdesc.dwFlags = DSBCAPS_STATIC |DSBCAPS_CTRLPAN
12     |DSBCAPS_CTRLVOLUME| DSBCAPS_GLOBALFOCUS;
13     dsdesc.dwBufferBytes = size;
14     dsdesc.lpwfxFormat = &swfmt;
15     result = pDS->CreateSoundBuffer( &dsdesc, &buf, NULL );
16     if(result != DS_OK)
```



```

17     MessageBox("建立次缓冲区失败!");
18     result = buf->Lock(0,size,&pAudio,&bytesAudio,NULL,NULL,NULL);
19     //锁定缓冲区
20     if(result != DS_OK)
21         MessageBox("锁定缓冲区失败!");
22     mmresult = mmioRead(hmmio,(HPSTR)pAudio,bytesAudio);
23     //读取声音文件资料
24     if(mmresult == -1)
25         MessageBox("读取声音文件资料失败!");
26     result = buf->Unlock(pAudio,bytesAudio,NULL,NULL);
27     //解除锁定缓冲区
28     if(result != DS_OK)
29         MessageBox("解除锁定缓冲区失败!");
30     mmioClose(hmmio,0);
31     return buf;
32 }

```

## 程序说明

(1) 第 1 行: 取得一个要打开的文件名称“filename”, 该函数的返回类型为 LPDIRECTSOUNDBUFFER。

(2) 第 3 行: 声明一个次缓冲区的对象指针。

(3) 第 4~7 行: 根据 filename 来打开文件, 取得 WAVE 文件格式与大小的程序代码不再列出, 读者可参考前面的内容。

(4) 第 9~17 行: 建立一个次缓冲区。

(5) 第 18~30 行: 将声音文件资料加载缓冲区中。

(6) 第 31 行: 返回前面建立的缓冲区对象。

由于要播放的声音文件有两个, 因此必须建立两个次缓冲区。如下所示。

## 程序代码

```

1  LPDIRECTSOUNDBUFFER pSBuf[2];           //声明次缓冲区
2  canvasFrame::canvasFrame()
3  {
4      hdc = ::CreateCompatibleDC(NULL);
5      Create(NULL,"绘图窗口",WS_POPUP);   //建立窗口
6      pSBuf[0] = createbuffer("bground.wav"); //加载背景音乐
7      pSBuf[1] = createbuffer("foot.wav");   //加载脚步声
8      pSBuf[0]->Play(0,0,1);               //播放背景音乐
9  }

```

## 程序说明

(1) 第 1 行: 声明了一个大小为 2 的次缓冲区数组。

(2) 第 6、7 行: 分别调用 createbuffer 函数并加载要打开的声音文件, 建立存储声音信息的次缓冲区。

(3) 第 8 行: 循环播放 pSBuf[0]缓冲区的声音, 即背景音乐。

OnTimer 函数中的程序代码以翻页的方式产生动画的效果。如下所示。

## 程序代码

```

1 void canvasFrame::OnTimer(UINT nIDEvent)
2 {
3     CFrameWnd::OnTimer(nIDEvent);
4     if(i==7)
5         i=0;
6     if(i==1 || i==4)                //判断图号
7         pSBuf[1]->Play(0,0,0);      //播放脚步声
8     /*贴图产生动画的程序代码*/
9
10 }

```

## 程序说明

第 6、7 行：在显示图号为 1 或 4 时，播放 pSBuf[1] 中的声音，即人物走动的脚步声。

## 运行结果

程序运行的结果如图 9-8 所示。



图 9-8

### 9.2.3 控制声音功能

DirectSound 中所有的声音控制与播放功能，都是在建立次缓冲区时赋予它的功能。

在前面内容里讨论建立次缓冲区方式时，在缓冲区的设定标志位中定义了“DSBCAPS\_CTRLPAN”与“DSBCAPS\_CTRLVOLUME”两个项目，使得缓冲区具有控制音量大小与声道播放的功能。事实上，DirectSound 也提供了一些控制声音的方法，这一小节中将参照范例 ch9\_2 学习声音控制的方法。

#### 1. 调整音量大小

调整音量大小的方法为“SetVolume”，使用语法如下：

```

HRESULT SetVolume(
    LONG lVolume        //音量大小
);

```

上述方法中输入的参数“lVolume”是介于 0~10000 之间的数值，负值越大表示声音越小。SetVolume 方法并没有放大音量的功能，只能把音量往下调，如果需要音量较大的声音文件就必须自己使用音效编辑软件制作。另外，在声音文件加载到次缓冲区时已是最大的音量。

## 2. 调整播放声道

控制播放声道，比较常见的就是控制左、右声道的声音大小，连续切换左右声道声音的播放，还可产生立体音的效果，DirectSound 中调整播放声道的方法为“SetPan”，其语法为：

```
HRESULT SetPan(
    LONG lPan
);
```

其中输入的参数“lPan”的值在-10000~10000 之间，其中负值越大代表左声道的声音越大，正值越大则代表右声道的声音越大，若值为 0 则表示两声道的音量相等。

下面再看一个范例程序 ch9-2（见随书光盘），这个程序可以让操作者选择要播放的声音文件，还可以调整音量大小与播放声道，其操作界面如图 9-9 所示。



图 9-9

在 OnCreate 函数中初始化 DirectSound 设定，并在窗口上配置要用的控件。下面省略一些前面已经出现过的内容，只对比较重要的程序部分进行说明。

### 程序代码

```
1 int canvasFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
2 {
3     if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
4         return -1;
5     //建立DirectSound的程序代码(略)
6     //建立文字卷标的程序代码(略)
7     listBox = new CListBox;
8     listBox->CreateEx(WS_EX_CLIENTEDGE, "ListBox",
9         NULL, LBS_STANDARD | WS_VISIBLE | WS_CHILD,
10         CRect(10, 50, 240, 270), this, 1); //建立 CListBox 对象
11     CString item[6] = {"宝贝奇想曲", "魔力狂飙五十音",
12         "新无敌炸弹超人", "巴冷公主", "陆战英豪", "决战时空"};
13     //选项名称数组
```

```

14  DWORD num[6] = {0,1,2,3,4,5};           //选项的数据值
15  int index;
16  char filename[10];
17  for(int i=0;i<=5;i++)
18  {
19      index = listbox->AddString(item[i]);    //加入各个选项
20      listbox->SetItemData(index,num[i]);     //设定各个选项的数据值
21      sprintf(filename,"s%d.wav",i);         //取得文件名称
22      pSBuf[i] = createbuffer(filename);     //加载背景音乐
23  }
24  listbox->SelectString(0,"巴冷公主");       //设定预设选项
25  stop = new CButton;
26  stop->Create("停止",BS_PUSHBUTTON|WS_VISIBLE,
27  CRect(10,280,80,310),this,2);            //建立停止按钮
28  play = new CButton;
29  play->Create("播放",BS_PUSHBUTTON|WS_VISIBLE,
30  CRect(90,280,160,310),this,3);           //建立播放按钮
31  loop = new CButton;
32  loop->Create("循环",BS_PUSHBUTTON|WS_VISIBLE,
33  CRect(170,280,240,310),this,4);          //建立循环按钮
34  CSliderCtrl *volumn = new CSliderCtrl;    //控制音量滑轨
35  volumn->Create(WS_VISIBLE|TBS_VERT,CRect(250,70,290,240),this,5);
36  volumn->SetRange(-2000,0,true);           //设定音量滑轨的范围
37  volumn->SetPos(-1000);                    //设定滑块位置
38  CSliderCtrl *channel = new CSliderCtrl;   //控制声道滑轨
39  channel->Create(WS_VISIBLE|TBS_VERT,CRect(290,70,330,240),this,6);
40  channel->SetRange(-10000,10000,true);     //设定声道滑轨的范围
41  channel->SetPos(0);                       //设定滑块位置
42  return 0;
43  }

```

#### 程序说明

- (1) 第7~23行：建立窗口中的 CListBox 控件，并在其中加入各个音乐选择的项目。
- (2) 第19行：组成声音文件名的字符串。
- (3) 第22行：调用前一个范例中看过的 createbuffer 自定义函数，建立次缓冲区并加载各个声音文件。
- (4) 第24~33行：建立【播放】、【停止】与【循环】按钮。各个窗口控件的建立是通过调用“Create”或“CreateEx”函数，并且在最后的一个参数中输入一个识别码。
- (5) 第34、35行：建立垂直的 CSliderCtrl 控件控制音量的大小。第36行的程序代码设定该滑轨的数值范围为0~-2 000；第37行的程序代码设定滑块开始的位置为-1 000，即中央位置。
- (6) 第38~41行：以同样的方法建立一个控制播放声道的 CSliderCtrl 控件，数值范围为-10 000~10 000。

接下来再看消息映射表中的内容：

#### 程序代码

```

1  BEGIN_MESSAGE_MAP(canvasFrame, CFrameWnd)
2  //{AFX_MSG_MAP(canvasFrame)

```

```

3  ON_WM_CREATE()
4  ON_BN_CLICKED(3,OnPlayDown)
5  ON_BN_CLICKED(4,OnLoopDown)
6  ON_BN_CLICKED(2,OnStopDown)
7  ON_WM_VSCROLL()
8  //}}AFX_MSG_MAP
9  END_MESSAGE_MAP()

```

## 程序说明

(1) 第4~6行：设定【播放】、【循环】与【停止】按钮被按下的消息处理函数为“OnPlayDown”、“OnLoopDown”和“OnStopDown”，处理函数按各个按钮的识别码来设定。

(2) 第7行：使用 Class Wizard 在程序中自动加入滚动条或滑轴滚动消息 WM\_VSCROLL 的处理函数。

接下来再编写一个使用者按下【播放】按钮时运行的处理函数，内容说明如下所示。

## 程序代码

```

1  void canvasFrame::OnPlayDown()
2  {
3      pSBuf[now]->Stop();
4      DWORD num ;
5      int index;
6      index = listbox->GetCurSel();           //选中选项索引值
7      num = listbox->GetItemData(index);       //取得选项数据值
8      pSBuf[num]->SetCurrentPosition(0);      //设定播放起点
9      pSBuf[num]->SetVolume(volume);          //设定音量
10     pSBuf[num]->SetPan(pan);                 //设定声道
11     pSBuf[num]->Play(0,0,0);                 //播放
12     now = num;                               //重设播放编号
13 }

```

## 程序说明

(1) 第3行：停止目前正在播放的音乐。

(2) 第6、7行：取得使用者选择的要播放音乐的编号“num”。

(3) 第8~10行：根据 num 值选择要播放的次缓冲区，以及播放的起点(0)、音量(volume)与声道(pan)，其中 volume 和 pan 为全局变量，它们的值随着使用者拖动滑轨而改变。

以下是使用者按下【循环】按钮时所运行的函数。

## 程序代码

```

1  void canvasFrame::OnLoopDown()
2  {
3      pSBuf[now]->Stop();
4      DWORD num ;
5      int index;
6      index = listbox->GetCurSel();           //选择选项索引值
7      num = listbox->GetItemData(index);       //取得选项数据值
8      pSBuf[num]->SetCurrentPosition(0);      //设定播放起点
9      pSBuf[num]->SetVolume(volume);          //设定音量
10     pSBuf[num]->SetPan(pan);                 //设定声道

```

```

11 pSBuf[num]->Play(0,0,1);           //循环播放
12 now = num;                          //重设播放曲号
13 }

```

**程序说明**

第 11 行：播放选择音乐时，若第 3 个参数为 1，则进行循环播放。

使用者按下【停止】按钮运行的函数如下所示。

**程序代码**

```

void canvasFrame::OnStopDown()
{
    pSBuf[now]->Stop();                //停止目前正在播放的音乐
}

```

**程序说明**

调用 Stop 方法停止目前正在播放的音乐。

OnVScroll 是处理 WM\_VSCROLL 消息的预设函数，参数“nSBCode”为滚动消息的代码：“nPos”为滑轨所在的位置；“pScrollBar”为产生消息的滑轨控件，下面来看看程序的内容说明。

**程序代码**

```

1 void canvasFrame::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
2 {
3     switch(pScrollBar->GetDlgCtrlID())    //取得滑轨控件的识别码
4     {
5         case 5:                          //滚动音量控制滑轨
6             if(nPos != 0)
7             {
8                 volume = nPos;           //设定音量值
9                 pSBuf[now]->SetVolume(nPos); //设定音量
10            }
11            break;
12        case 6:                          //滚动声道控制滑轨
13            if(nPos != 0)
14            {
15                pan = nPos;               //设定声道值
16                pSBuf[now]->SetPan(nPos); //设定声道
17            }
18            break;
19        }
20    CFrameWnd::OnVScroll(nSBCode, nPos, pScrollBar);
21 }

```

**程序说明**

- (1) 第 3 行：取得发出消息的滑轨控件识别码，switch 判断是哪一个滑轨发出的消息。
- (2) 第 5 行：取得识别码为“5”，表示使用者拖动了调整音量的滑轨。
- (3) 第 8、9 行：分别重设音量变量 volume 的值以及播放缓冲区的音量。
- (4) 第 12 行：取得识别码为“6”，表示使用者拖动了调整声道的滑轨。
- (5) 第 15、16 行：重设声道变量 pan 的值以及左右声道声音播放的大小。

通过以上的设定，当使用者拖动滑轨，播放音乐的音量与声道便会随之改变。

## 运行结果

程序运行的结果如图 9-10 所示。

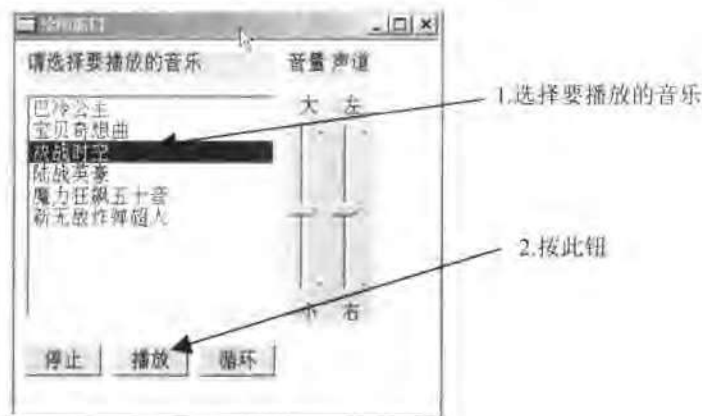


图 9-10

## 9.3 3D 音效的实际演练

DirectX 的最大特色在于它在 Windows 平台上以全动态、实时 3D 的技术来达到最大的声光效果，Direct Graphics 就是这些新技术中全 3D 化最明显的部分，另外 DirectSound 也提供了一个身临其境的环境音效空间。

3D 音效与 2D 音效的不同之处主要在于它加入了空间感。例如，在自己的左边有只小狗在叫，那么很容易听出声音由左方传来，如果这只小狗边叫边奔跑着朝自己所在的方向而来，那么听到的狗吠声将会越来越大。

为了产生 3D 的声音效果，DirectSound 提供了方法与接口来模拟真实世界中声音的发出与倾听过程，在本节中将介绍制作以及在程序中加入 3D 音效的方法。

### 9.3.1 认识 3D 音效

3D 音效的概念是以“发声者”与“倾听者”为基础，这两者间存在着一些影响声音效果的因素，因此形成具有空间感的 3D 音效。本小节里将说明几个与 3D 声音相关的特性，这些特性在 DirectSound 中都有设定的方法，在程序中按照发声者与倾听者之间的相互关系来定义这些特性以产生具有变化的 3D 音效。

#### 1. 最大距离与最小距离

最大距离与最小距离是属于发声者的特性，当倾听者越来越靠近发声者时，听到的声音会越来越大，而当靠近到某一距离之内时，听到的声音就不再继续增大，这段距离就是“最小距离”。

当倾听者渐渐远离发声者时，听到的声音会越来越小，当在某一距离之外时，倾听者所听到的声音就不再继续变小，这段距离也就是“最大距离”。下面以图 9-11 来说明最小距离与最大距离的

意义。



图 9-11

## 2. 位置

位置即为发声者与倾听者在三维空间中的所在位置。随着两者的相对位置不同，倾听者听到的声音效果也不同。

发声者与倾听者的所在位置，是以三维空间的向量表示的，相对于其在三维空间中的点坐标  $(x,y,z)$  而言，其向量表示为  $(x,y,z)$ 。

## 3. 速度

速度为发声者或倾听者在三维空间中的移动速度，该特性同样会改变两者在空间中的坐标，产生不同的声音效果。

## 4. 发声者声音锥

在预设的情况下，DirectSound 中的声音是以“点声音源”向四周发散，即倾听者所听到的声音只受到与发声者相对距离的影响。

在 DirectSound 中可设定发声者的“声音锥”，声音锥是一个有角度与方向性的圆锥，倾听者所听到的声音效果会受到声音锥的影响，图 9-12 是一个声音锥的示意图。

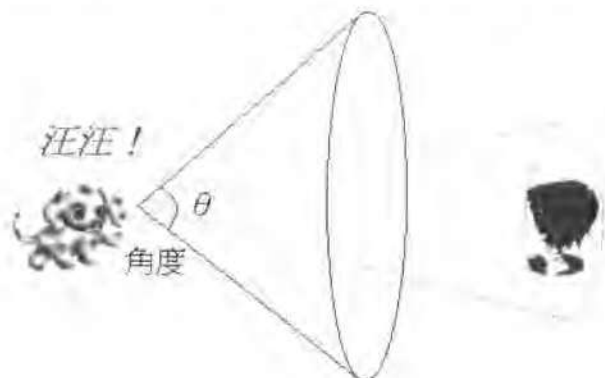


图 9-12

从上图中可看出声音锥的几个特点：

- (1) 分为内外两层。
- (2) 有一个方向性。
- (3) 具有一定角度。



在内层声音锥里，倾听者所听到的声音是受到与发声者距离长度的影响，而在内外层声音锥之间则为一个转换区，声音的大小会受一个转换系数的影响而变化；如果在外层声音锥外，则声音会根据位置远近而变化。

## 5. 倾听者面对方向

倾听者所面对的方向也会影响其所听到的声音效果，在 DirectSound 中以两个向量来决定倾听者面对的方向：一为倾听者头顶的向量，一为倾听者面对方向的向量，如图 9-13 所示。



图 9-13

在清楚了 DirectSound 中影响 3D 声音效果的一些主要特性后，接下来的内容将讨论如何在程序中建立发声者与倾听者，并设定两者的特性以产生 3D 音效。

## 9.3.2 建立倾听者功能

倾听者是一个听到声音的对象。事实上，这个对象所听到的声音便是使用者所听到的声音。要建立倾听者，必须由主缓冲区取得 IDirectSound3DListener 接口并在主缓冲区的设定中加入“DSBCAPS\_CTRL3D”标志位，使其具有 3D 音效的控制功能。下面来看看建立倾听者的程序代码及说明。

### 程序代码

```

1  LPDIRECTSOUNDBUFFER pPBuf;           //声明主缓冲区指针
2  LPDIRECTSOUND3DLISTENER Listener;    //声明倾听者指针
3  memset( &dsdesc, 0, sizeof(dsdesc) );
4  dsdesc.dwSize = sizeof(dsdesc);
5  dsdesc.dwFlags = DSBCAPS_PRIMARYBUFFER|DSBCAPS_CTRL3D;
6  dsdesc.dwBufferBytes = 0;
7  dsdesc.lpwfxFormat = NULL;
8  result = pDS->CreateSoundBuffer( &dsdesc, &pPBuf, NULL );
9  if(result != DS_OK)
10  MessageBox("建立主缓冲区失败!");
11  result = pPBuf->QueryInterface( IID_IDirectSound3DListener,
12  (VOID**)&Listener );
13  if(result != DS_OK)
14  MessageBox("建立倾听者失败!");
    
```

```
15 pPBuf->Release();
```

#### 程序说明

- (1) 第 2 行: 声明一个 LPDIRECTSOUND3DLISTENER 的倾听者指针“Listener”。
- (2) 第 5 行: 设定缓冲区特性时, 加入“DSBCAPS\_CTRL3D”标志位, 使其具有 3D 控制的功能。
- (3) 第 11、12 行: 建立主缓冲区“pPBuf”后, 调用 QueryInterface 方法取得 LPDIRECTSOUND-3DLISTENER 接口并建立倾听者。
- 如此一来, 便建立了一个倾听者对象“Listener”, 接下来可使用 LPDIRECTSOUND3DLISTENER 接口下的各种方法来设定倾听者的特性, 常用的方法见表 9-6。

表 9-6

方 法	说 明
SetPosition	设定倾听者的位置
GetPosition	取得倾听者的位置
SetVelocity	设定倾听者的速度
GetVelocity	取得倾听者的速度
SetOrientation	设定倾听者面对的方向
GetOrientation	取得倾听者面对的方向
SetAllParameters	设定倾听者的全部特性
GetAllParameters	取得倾听者的全部特性

至此已经建立一个倾听者来“听”声音了, 不过这时还只是一个寂静的世界, 因为程序中还没有任何会发出声音的发声者, 下一小节将介绍在程序中建立发声者的方式。

### 9.3.3 建立发声者

发声者是一个会发出声音的对象, 在 DirectSound 中建立一个发声者就等于建立一个具有 3D 音效控制能力的次缓冲区, 而且这个次缓冲区必须取得 LPDIRECTSOUND3DBUFFER 接口。相对于程序中只能有一个倾听者 (因为只有一个使用者), 发声者却可以有多个。

在 DirectSound 中, 每一个具有 3D 功能的次缓冲区 (发声者), 都有其对应的一个“LPDIRECTSOUND3DBUFFER”类型的 3D 缓冲区用来运行 3D 音效的功能。下面对以下建立次缓冲以及 3D 缓冲区的程序内容进行说明。

#### 程序代码

```
1  LPDIRECTSOUNDBUFFER  pSBuf;           //声明次缓冲区
2  LPDIRECTSOUND3DBUFFER Buffer3D;       //声明 3D 缓冲区指针
3  memset( &dsdesc,0,sizeof(dsdesc));
4  dsdesc.dwSize = sizeof(dsdesc);
5  dsdesc.dwFlags = DSBCAPS_STATIC | DSBCAPS_CTRLPAN |
6  DSBCAPS_CTRLVOLUME | DSBCAPS_GLOBALFOCUS |
7  DSBCAPS_CTRL3D;
8  dsdesc.dwBufferBytes = size;
9  dsdesc.lpwfxFormat = &swfmt;
10 result = pDS->CreateSoundBuffer( &dsdesc, &pSBuf, NULL );
```

```

11 if(result != DS_OK)
12     MessageBox("建立次缓冲区失败!");
13 result = pSBuf->QueryInterface( IID_IDirectSound3DBuffer,
14     (VOID**)&Buffer3D );
15 if(result != DS_OK)
16     MessageBox("建立 3D 缓冲区失败!");

```

## 程序说明

- (1) 第 2 行：声明一个 LPDIRECTSOUND3DBUFFER 的 3D 缓冲区指针“Buffer3D”。
- (2) 第 7 行：在设定缓冲区特性时，加入“DSBCAPS\_CTRL3D”标志位，使其具有 3D 控制的功能。
- (3) 第 10 行：建立次缓冲区“pSBuf”。
- (4) 第 13 行：调用 QueryInterface 方法来取得 LPDIRECTSOUND3DLISTENER 接口并建立该次缓冲区的 3D 缓冲区。

在建立了次缓冲区以及对应的 3D 缓冲区后，接下来只要在该次缓冲区中加载声音文件，并调用 Play 方法来播放，DirectSound 便会按照倾听者与发声者之间的关系来产生倾听者所听到的 3D 音效。表 9-7 列出了设定发声者特性的主要方法。

表 9-7

方 法	说 明
SetMaxDistance	设定最大距离
GetMaxDistance	取得最大距离
SetMinDistance	设定最小距离
GetMinDistance	取得最小距离
SetPosition	设定发声者的位置
GetPosition	取得发声者的位置
SetVelocity	设定发声者的速度
GetVelocity	取得发声者的速度
SetConeAngles	设定声音锥的角度
GetConeAngles	取得声音锥的角度
SetConeOrientation	设定声音锥的方向
GetConeOrientation	取得声音锥的方向
SetConeOutsideVolume	设定声音锥外声音的大小
GetConeOutsideVolume	取得声音锥外声音的大小
SetAllParameters	设定发声者的全部特性
GetAllParameters	取得发声者的全部特性

次缓冲区只负责声音的播放控制，而其对应的 3D 缓冲区可以调用上述方法来产生 3D 音效。

接下来看一个范例 ch9\_3（见随书光盘），在这个范例中将固定发声者的位置，而使用者可以由鼠标控制屏幕上的人物（代表声音倾听者）来移动其位置，如此一来将会听到不同声音所发出的 3D 效果。

在这个范例里，在 OnCreate 函数中建立了倾听者“Listener”与发声者“Buffer3D”并加载要播放的声音。

## 程序代码

```

1  LPDIRECTSOUND3DBUFFER  Buffer3D;
2  LPDIRECTSOUND3DLISTENER Listener;
3  float xvector,yvector;
4  canvasFrame::canvasFrame()
5  {
6  /*建立窗口与加载图文件的程序代码(略)*/
7  CPoint *p = new CPoint(250,150);           //设定起始坐标
8  CClientToScreen(p);                         //转换坐标
9  ::SetCursorPos(p->x,p->y);                  //设定鼠标所在位置
10 ShowCursor(false);
11 pSBuf->Play(0,0,1);
12 }

```

## 程序说明

(1) 第 7 行：设定窗口中人物开始的显示位置。

(2) 第 11 行：循环播放音乐。

要注意的一点是，屏幕上人物所在位置的坐标与声音倾听者的位置是没有关系的，不过程序中会利用人物的所在坐标来计算并重设倾听者的位置。

## 程序代码

```

1  void canvasFrame::OnMouseMove(UINT nFlags, CPoint point)
2  {
3      CClientDC dc(this);
4      xlast = xnow;                               //上一次的 X 坐标
5      ylast = ynow;                               //上一次的 Y 坐标
6      xnow = point.x;                             //此次的 X 坐标
7      ynow = point.y;                             //此次的 Y 坐标
8      /*控制人物移动的程序代码(略)*/
9      xvector = (xnow-250)/50*0.5;                //计算倾听者的 X 坐标
10     yvector = (ynow-150)/50*0.5;                //计算倾听者的 Y 坐标
11     Listener->SetPosition(xvector,yvector,0,DS3D_IMMEDIATE);
12     char str[40];
13     sprintf(str,"目前倾听者的位置: (%.1f,%.1f,0) ",xvector,yvector);
14     dc.TextOut(0,0,str);
15     CFrameWnd::OnMouseMove(nFlags, point);
16 }

```

## 程序说明

当使用者移动鼠标控制窗口中人物的移动时，运行上面的这个 OnMouseMove 函数。这里提供一个新的构想，就是以前面所设的原点为参考点，且计算人物移动后在 X、Y 轴方向距原点的距离，并根据计算后的结果来设定倾听者的位置，下面以图 9-14 做说明。

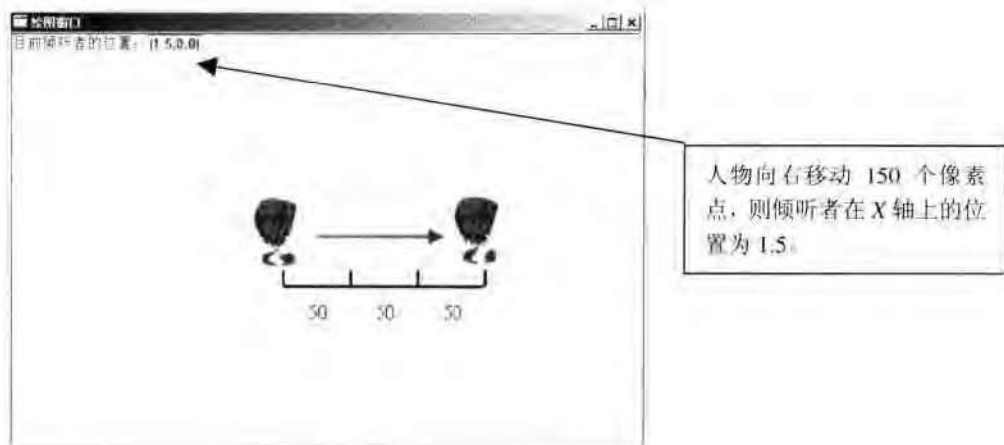


图 9-14

上图中人物每在 X 或 Y 轴方向移动 50 个像素点, 则倾听者在 X 或 Y 轴方向的位置便增加 0.5。

- (1) 第 9、10 行: 计算人物移动后倾听者的所在坐标。
- (2) 第 11 行: 调用 `SetPosition` 函数设定倾听者的位置, 其中输入的各个参数的说明见表 9-8。

表 9-8

参 数	说 明
<code>xvector</code>	倾听者在 X 轴方向的坐标
<code>yvector</code>	倾听者在 Y 轴方向的坐标
<code>0</code>	倾听者在 Z 轴方向的坐标
<code>DS3D_IMMEDIATE</code>	设定此参数则音效会依据设定值立即改变。另一参数为 <code>DS3D_DEFERRED</code> , 若设定此参数, 则在调用 <code>CommitDeferredSettings</code> 函数之前, 声音的效果将不会依据设定值而产生变化

- (3) 第 13、14 行: 组成字符串, 在窗口中显示倾听者的坐标位置。

## 运行结果

程序运行结果如图 9-15 所示。



图 9-15

当运行这个范例的时候，可能会发现人物在坐标右边时，左边的喇叭会播放声音，表示声音由左边传来而当人物距离原点越远时，声音则会变得越小。

在这一章里对使用 DirectSound 在程序中加入音效的方法做了完整的介绍，而对音效的播放与控制，相信也是各位学习游戏设计必修的重要课程。

## 课后重点整理

- DirectSound 提供了对各种音效处理的支持，如低延迟音、3D 立体音、协调硬件运作等等音效功能。
- 建立 DirectSound 的第一个步骤是必须确认在 VC++ 中是否已经设定好引用 DirectX 头文件与函数库的所在路径。
- Direct Graphics 中是使用“影像材质”(Texture)来显示图片，而在 DirectSound 中则是使用“声音缓冲区”来播放声音。
- 在 DirectSound 程序中至少要有有一个次缓冲区来存储要播放的声音。次缓冲区的建立必须要参考所加载声音档(.wav)文件的两种信息：“声音文件格式”与“波形资料大小”。
- WAVE 声音文件是符合 RIFF (Resource Interchange File Format) 规格的一种多媒体文件，RIFF 规格的文件是利用“区块(chunk)”的方式来存储文件，包含记录文件格式的“格式区块(fmt)”与文件实际内容的“资料区块(data)”。
- 混音就是将多种声音的波形组合形成一个新的波形。
- 在 DirectSound 中可以同时播放数个声音，混音的操作则是由 DirectSound 将各个次缓冲区的声音资料复制到主缓冲区中并自动进行混音的工作，最后再将其输出。
- 3D 音效与 2D 音效的不同之处主要在于它加入了空间感。
- 3D 音效的概念是以“发声者”与“倾听者”为基础，这两者之间存在着一些影响声音效果的因素，因此形成具有空间感的 3D 音效。
- 在预设的情况之下，DirectSound 中的声音是以“点声音源”向四周发散，即倾听者所听到的声音只受到与发声者相对距离的影响。

### 课后习题

1. 建立 DirectSound 系统，必须经过哪些步骤？
2. SetCooperativeLevel (m\_hWnd, DSSCL\_PRIORITY) 是用来设定协调层级，其中参数 1 设为“m\_hWnd”，表示为应用程序主窗口，参数 2 为设定标志符(flag)，设定程序使用资源的优先权。如表 9-9 所示，请填入所设定的标志符。

表 9-9

设定标志符	说 明	缺 点
	独占模式，当程序为作用时，则只有此程序可使用播放声音的资源	背景应用程序都处于静音状态
	正常模式，播放声音的资源可与其他程序共享	无法改变 DirectSound 主要格式
	可设定主缓冲区的播放模式	可能改变其他软件的输出格式
	具有最高优先权，可直接存取主缓冲区的内容	辅助缓冲区将无法播放，其他应用软件会失去所属的暂存区

3. 请简单说明 DirectSound 中所使用的两种缓冲区类型。
4. 试简述 VC++ 中打开与加载 WAVE 文件的流程和步骤。
5. 请试着说明什么是最大距离与最小距离？
6. 从声音锥的示意图 9-16 中可看出声音锥有哪些特点？试简述之。

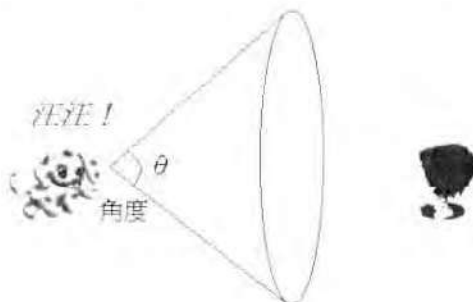


图 9-16

## 第 10 章 DirectInput 的使用方法

游戏和电影最大的差别就在于游戏提供给使用者进行“互动”的平台。从技术上的角度来说，软件工程师可以通过接收的 Windows 消息取得玩家操作的进度，或调用 `GetKeyboardState` 函数取得目前键盘状态。软件工程师还可以使用 Windows 提供的消息或 API 函数来取得使用者的输入状态，但使用 `DirectInput` 可以处理键盘或鼠标消息提供的服务。从另一角度而言，`DirectX` 以“直接”与硬件进行交流闻名，而非等待 Windows 传送消息，`DirectInput` 可以让使用者直接取得消息，立即响应目前的硬件状况。

### 10.1 建立 DirectInput 程序

在前面章节中已经讨论了一些在 VC++ 中的消息处理技巧以及使用者输入的方法，那些是设计 VC++ 窗口程序必须学会的基础知识。在本章中将进一步探讨 `DirectX` 中专门处理使用者输入消息的组件“`DirectInput`”。

`DirectInput` 在游戏程序中或许比不上 `Direct Graphics` 或 `DirectSound` 那样可以制作出华丽的画面或动人的音乐特效，但它却是制作游戏不可或缺的角色。凡是键盘、鼠标、摇杆或者力回馈装置的使用，都必须借助它的功能来辅助完成，而 `DirectInput` 本身的功能则可以帮助程序设计者轻易地将各类输入装置消息的工作加以简化。

#### 10.1.1 开始建立 DirectInput 程序

在使用 `DirectInput` 的程序前，必须先确认是否已经设定好引用 `DirectX` 的头文件与函数库。接着按照图 10-1、图 10-2、图 10-3 中的步骤来设定连接时需要使用的函数库与头文件。

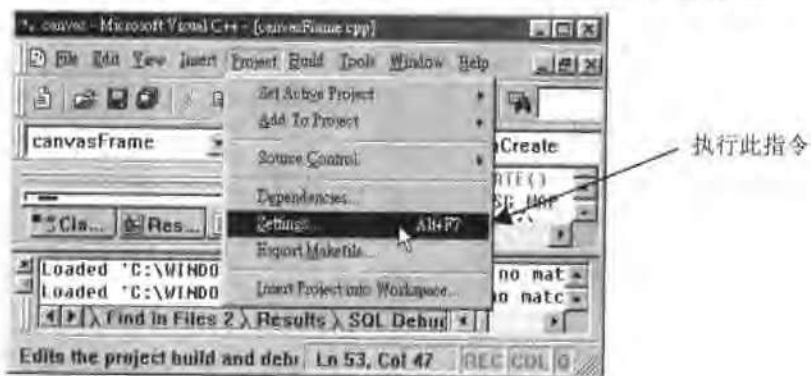


图 10-1



输入“Dxguid.lib”与“Dinput.lib”  
编译连接文件

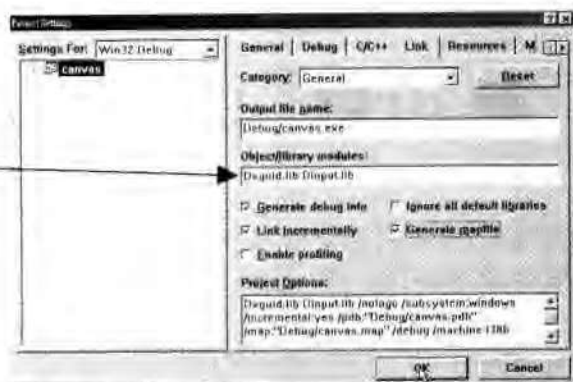
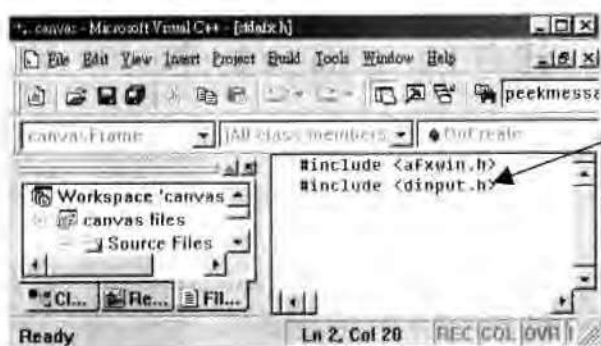


图 10-2



输入引用“dinput.h”的头文件

图 10-3

在完成了连接函数库与头文件的设定后，按照下面的内容一步步地学习使用 DirectInput。

## 10.1.2 建立 DirectInput 对象

无论使用什么装置作为输入的来源，首先都必须建立 DirectSound 对象，如下所示。

### 程序代码

```

1  LPDIRECTINPUT7 pDI;           //声明 DirectInput 对象指针
2  HRESULT result;               //声明 HRESULT 类型变量
3  HINSTANCE hinst = AfxGetInstanceHandle(); //取得应用程序的 handle
4  result = DirectInputCreateEx(hinst, DIRECTINPUT_VERSION,
5  IID_IDirectInput7, (void*)&pDI, NULL); //建立 DirectInput 对象
6  if(result != DI_OK)
7  MessageBox("建立 DirectInput 对象失败!");
  
```

### 程序说明

- (1) 第 1 行：声明一个 LPDIRECTINPUT7 类型的 DirectInput 的对象指针“pDI”。
- (2) 第 3 行：建立 DirectInput 对象需要的应用程序的 handle，并运行“AfxGetInstanceHandle”函数来取得应用程序的 handle。
- (3) 第 4 行：调用“DirectInputCreateEx”来建立 DirectInput 对象，输入的各个参数及其意义如表 10-1 所示。

表 10-1

参 数	说 明
hinst	参数 1 必须输入应用程序的 handle
DIRECTINPUT_VERSION	参数 2 输入代表 DirectInput 版本的代码
IID_IDirectInput7	参数 3 必须设为 IID_IDirectInput7
(void**)&pDI	参数 4 须输入 DirectInput 对象的指针, 并转换为 LPVOID 类型
NULL	参数 5 必须设为 NULL

(4) 第 6 行: 判断返回值 result, 若等于“DI\_OK”则表示建立对象成功, 否则显示错误消息。

### 10.1.3 建立输入装置对象

在建立了 DirectInput 对象之后, 接下来必须建立起要使用的输入装置, 如下面的程序代码则是建立一个“键盘”为输入的装置对象。

#### 程序代码

```

1  LPDIRECTINPUTDEVICE7 pDKB;           //声明输入装置对象指针
2  result = pDI->CreateDeviceEx(GUID_SysKeyboard, IID_IDirectInputDevice7,
3      (void**)&pDKB, NULL);           //建立输入装置对象
4  if(result != DI_OK)
5      MessageBox("建立键盘输入装置失败!");

```

#### 程序说明

(1) 第 2 行: 运行“CreateDeviceEx”方法建立输入装置对象“pDKB”, 表示程序把键盘当作输入设备, 其中输入的各个参数其说明见表 10-2。

表 10-2

参 数	说 明
GUID_SysKeyboard	参数 1 输入要建立输入装置的识别码, 由于使用键盘当作输入设备, 在此必须设为 GUID_SysKeyboard
IID_IDirectInputDevice7	参数 2 输入使用接口的识别码, 在此设为 IID_IDirectInputDevice7
(void**)&pDKB	参数 3 须输入建立对象的指针 pDKB, 并转换为 LPVOID 类型
NULL	参数 4 须设为 NULL

(2) 第 4 行: 判断返回值, 若为“DI\_OK”表示建立输入装置对象成功。

### 10.1.4 资料格式的设定

在建立需要使用的输入装置后, 接下来必须设定该装置的资料格式。因为每一种装置资料输入与读取的方式并不一样, 例如鼠标为滚动, 键盘则为按下与放开。下面的程序代码接着前面使用键盘的例子, 设定键盘输入装置的的资料格式。

#### 程序代码

```

1  result = pDKB->SetDataFormat(&c_dfDIKeyboard);

```

```
2  if(result != DI_OK)
3  MessageBox("设定资料格式失败!");
```

## 程序说明

第 1 行：调用“SetDataFormat”方法来设定资料格式，输入的参数为“&c\_dfDIKeyboard”表示键盘，若是鼠标或者摇杆，则输入的参数分别为“&c\_dfDIMouse”与“&c\_dfDIJoystick”。

## 10.1.5 设定程序协调层级

设定 DirectInput 的使用协调层级的程序内容如下所示。

## 程序代码

```
1  result = pDKB->SetCooperativeLevel(m_hwnd, DISCL_BACKGROUND |
2  DISCL_NONEXCLUSIVE);           //设定协调层级
3  if(result != DI_OK)
4  MessageBox("设定程序协调层级失败!");
```

## 程序说明

“LPDIRECTINPUTDEVICE7”接口下同样有个“SetCooperativeLevel”方法用来设定程序的协调层级，上面第一行程序代码便是调用此方法设定键盘输入装置协调层级的，其中输入的第 1 个参数为程序窗口的 handle “m\_hwnd”，第 2 个参数是设定协调层级的特性，此参数可设定的值与说明见表 10-3。

表 10-3

设定参数	说明
DISCL_BACKGROUND	程序可在非作用中时，取得输入装置的输入资料
DISCL_EXCLUSIVE	独占模式，其他程序将无法使用程序所建立与使用的输入装置
DISCL_FOREGROUND	程序只有在作用中时，才可取得输入装置的输入资料
DISCL_NONEXCLUSIVE	非独占模式，其他程序可与程序共享输入装置
DISCL_NOWINKEY	无法使用 Windows 按键，防止使用者按下中断键结束程序

## 10.1.6 输入装置的调用方法

通过前面的各项操作后，已经成功建立了所要使用的输入装置对象，并完成了各项特性的设定。然后必须由程序来“调用”这个已经定义好的输入装置，如下所示。

## 程序代码

```
1  result = pDKB->Acquire(); //调用输入装置
2  if(result != DI_OK)
3  MessageBox("调用输入装置失败!");
```

## 程序说明

第 1 行：调用“Acquire”的方法由程序取得键盘“pDKB”的控制权，即此时在键盘上按下任意键，程序便开始接收所输入的消息。

## 10.2 键盘与鼠标输入的取得方法

使用 DirectInput 取得键盘或者鼠标的输入方法基本上大同小异, 在 10.1 节建立 DirectInput 程序的流程步骤中, 在“建立输入装置对象”与“设定资料格式”时, 必须输入不同的参数来区分使用的输入装置是键盘或者鼠标。

这一节里来看看本节主要讲述如何运用 DirectInput 读取标准系统输入装置“键盘”与“鼠标”的输入消息

### 10.2.1 键盘输入的取得

在正式介绍如何取得键盘输入信息之前, 先讲一个重要概念, 即在一个必须取得使用者输入的程序中, 程序要不断地读取目前使用者在输入装置上的输入状态。只有这样, 才能在任一时刻都能检测到使用者输入的情况, 并立即做出适当的处理与响应, 下面就针对取得键盘输入的状态进行说明。

#### 1. 取得键盘输入状态

如果把键盘当做输入装置, 那么就必须先建立一个大小为 256 位的缓冲区, 用来暂存每个按键的状态 (按下或松开), 然后不断地调用“GetDeviceState”方法取键盘的状态并将其存入缓冲区中, 接着再判断缓冲区中的内容, 便可知道目前哪些按键是按下或松开的。下面来看一段读取键盘消息的程序代码。

##### 程序代码

```
1 char buffer[256]; //设定缓冲区
2 result = pDKB->GetDeviceState(sizeof(buffer), (LPVOID)&buffer);
3 if(result != DI_OK)
4     MessageBox("取得键盘状态失败!");
5 if(buffer[DIK_RIGHT] & 0x80)
6     /*按下右键所运行的程序代码*/
7 if(buffer[DIK_LEFT] & 0x80)
8     /*按下左键所运行的程序代码*/
9 if(buffer[DIK_UP] & 0x80)
10    /*按下上键所运行的程序代码*/
11 if(buffer[DIK_DOWN] & 0x80)
12    /*按下下键所运行的程序代码*/
```

##### 程序说明

(1) 第 1 行: 建立一个代表键盘输入状态的缓冲区数组, 大小为 256, 设为全局变量。

(2) 第 2~12 行: 检测键盘状态并响应的程序代码, 此段程序代码必须放在程序的“大循环”或者“OnTimer”函数里, 不断地被运行。其中在第 2 行中, 键盘装置对象 PKDB 调用“GetDeviceState”方法来取得目前键盘的状态并存入 buffer 缓冲区中。

## 2. 判断键盘输入状态

在上一个例子中，判断“右”键是否被按下的程序代码为：

```
if(buffer[DIK_RIGHT] & 0x80)
```

其中“DIK\_RIGHT”为预设的索引值，而键盘上每一个按键相对于缓冲区数组都有一个预设的索引值，例如：

```
buffer[DIK_LEFT]           //代表左键
buffer[DIK_UP]             //代表上键
buffer[DIK_DOWN]          //代表下键
```

而缓冲区中每个代表键盘的数组元素值必须将其与“0x80”做“&(And)”运算，返回值如果为“1”，则表示该按键处于被按下的状态，否则为松开的状态。

接下来创建一个范例 ch10\_1（见随书光盘），使用 DirectInput，并以键盘为输入装置，让使用者可以按下【↑】、【↓】、【←】、【→】键来控制窗口中小球的移动。

### 程序代码

```
1 LPDIRECTINPUT7 pDI;           //声明 DirectInput 对象指针
2 LPDIRECTINPUTDEVICE7 pDKB;    //声明输入装置对象指针
3 HRESULT result;               //声明 HRESULT 类型变量
4 char buffer[256];             //设定缓冲区
5 int x,y;                      //声明贴图坐标
6 canvasFrame::canvasFrame()
7 {
8     //建立窗口的程序代码(略)
9     mdc = new CDC;
10    mdc->CreateCompatibleDC(&dc);
11    ball = new CBitmap;
12    ball->m_hObject = (HBITMAP)::LoadImage(NULL,"ball.bmp",
13    IMAGE_BITMAP,81,81,LR_LOADFROMFILE); //加载图文件
14    mdc->SelectObject(ball);
15    x = ((rect.right-rect.left) - 81)/2; //设定小球起点 x 坐标
16    y = ((rect.bottom-rect.top) - 81)/2; //设定小球起点 y 坐标
17 }
```

### 程序说明

这个范例仅示范了 DirectInput 的使用，使用 DC 贴图的方式简化程序，下面是 canvasFrame 建构中的程序内容说明：

- (1) 第 1~4 行：声明 DirectInput 所需的对象指针与缓冲区。
- (2) 第 5 行：声明贴图的 x、y 坐标变量。
- (3) 第 8~13 行：建立屏幕贴图所需的 DC 并加载图文件。
- (4) 第 15、16 行：设定 x 与 y 坐标，使得开始时小球图片显示在操作窗口的正中央。

接下来初始化 DirectInput 的操作就在 OnCreate 的函数中，并且建立了一个定时器，发出 WM\_TIMER 消息的间隔必须越短越好（设为 0，让程序可以在任何时刻都能读取到使用者的输入状态）。下面来看看在 OnTimer 函数中取得键盘输入状态以及处理使用者输入的程序内容。

## 程序代码

```

1 void CanvasFrame::OnTimer(UINT nIDEvent)
2 {
3     CFrameWnd::OnTimer(nIDEvent);
4     CClientDC dc(this); //取得操作窗口 DC
5     dc.BitBlt(x,y,81,81,mdc,0,0,SRCCOPY); //贴图
6     result = pDKB->GetDeviceState(sizeof(buffer), (LPVOID)&buffer);
7     //取得键盘状态
8     if(result != DI_OK)
9         MessageBox("取得键盘状态失败!");
10    if(buffer[DIK_RIGHT] & 0x80) //判断右键是否被按下
11        if(x+80 > rect.right) //判断是否碰到右边缘
12            x = rect.right - 60;
13    else
14        x+=20;
15    if(buffer[DIK_LEFT] & 0x80) //判断左键是否被按下
16        if(x-20 < -21) //判断是否碰到左边缘
17            x = -21;
18    else
19        x-=20;
20    if(buffer[DIK_UP] & 0x80) //判断上键是否被按下
21        if(y-20 < -21) //判断是否碰到上边缘
22            y = -21;
23    else
24        y-=20;
25    if(buffer[DIK_DOWN] & 0x80) //判断下键是否被按下
26        if(y+80 > rect.bottom) //判断是否碰到下边缘
27            y = rect.bottom-60;
28    else
29        y+=20;
30 }

```

## 程序说明

(1) 第 6 行：取得目前键盘上的输入状态并存入缓冲区数组 buffer 中。

(2) 第 10、15、20、25 行：判断【↑】、【↓】、【←】、【→】键是否被按下，依按下的键来决定下一次小球图片贴图的位置。由于在判断下次贴图的坐标时，必须判断是否超过窗口边框（不然小球会移动到不见），因此必须知道小球图片的大小格式，如图 10-4 所示。

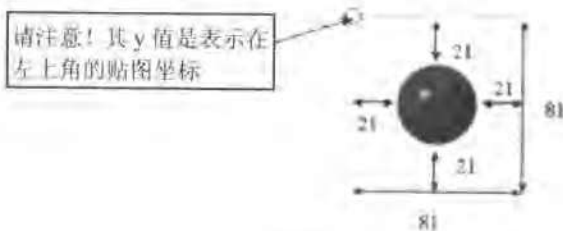


图 10-4

下面来看看这个范例的运行结果。

## 运行结果

程序运行结果如图 10-5 所示。

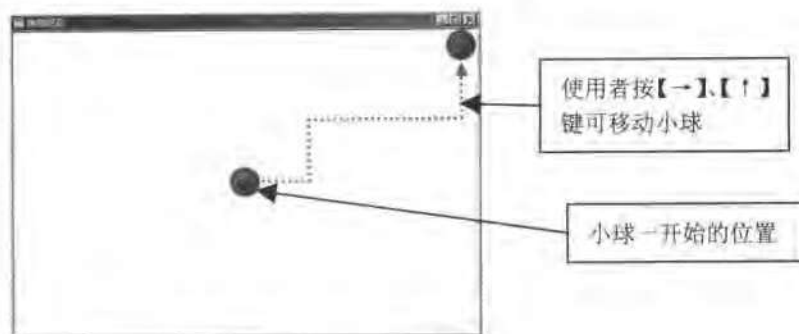


图 10-5

## 技巧

在游戏程序中，通常会使用所谓的“大循环”，这个循环会不断地运行程序，而程序本身可能包含显示画面、播放声音、检测使用者输入状态等操作。当产生了一个跳出程序的消息，例如使用者按下了程序的【结束】按钮，则结束程序。由图 10-6 可看出大循环的概念。

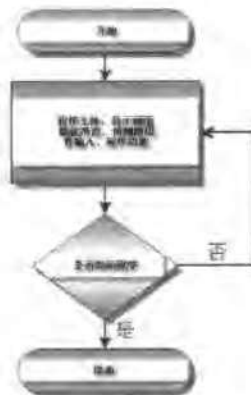


图 10-6

## 10.2.2 取得鼠标输入

鼠标的输入状态可分为两种：

- (1) 鼠标的移动；
- (2) 按下鼠标按钮。

虽然鼠标的输入状态和键盘并不相同，但是鼠标取得消息的概念却是和键盘一样的。这一小节里将讨论取得鼠标输入消息的方法。

### 1. 设定鼠标为输入装置

设定鼠标为输入装置的步骤与键盘大致相同，不过在建立输入装置对象与设定资料格式时必须输入不同的参数，代码如下所示：

**程序代码**

```

1  pDI->CreateDeviceEx(GUID_SysMouse, IID_IDirectInputDevice7,
2  (void**)&pDMO, NULL);           //建立输入装置对象
3  pD->SetDataFormat(&c_dfDIMouse2); //设定资料格式

```

**程序说明**

上述的程序代码中，调用“CreateDeviceEx”函数建立输入装置对象“pDMO”，其中第 1 个输入的参数必须设为“GUID\_SysMouse”，表示输入装置为鼠标；而设定资料格式则必须输入参数“&c\_dfDIMouse2”。

**2. DIMOUSESTATE2 结构**

有别于使用键盘时自定义一个大小为 256 位的缓冲区来暂存键盘状态，鼠标输入状态是存储在“DIMOUSESTATE2”结构中。下面是这个结构中所定义的内容：

```

typedef struct DIMOUSESTATE2 {
    LONG lX;
    LONG lY;
    LONG lZ;
    BYTE rgbButtons[8];
} DIMOUSESTATE2, *LPDIMOUSESTATE;

```

以下是对这个结构的内容说明：

- (1) lX：鼠标在 X 轴（水平）方向上的“移动量”，正数代表向右移，负数代表向左移。
- (2) lY：鼠标在 Y 轴（垂直）方向上的“移动量”，正数代表向下移，负数代表向上移。
- (3) lZ：鼠标在 Z 轴方向（有滚轮的鼠标）上的“移动量”，正数代表向后滚动，负数代表向前滚动。
- (4) rgbButtons[8]：存储各个按键的状态，若为两键鼠标，则 rgbButtons[0]代表左键，rgbButtons[1]代表右键。若有其他按键则依次类推。

在范例 ch10\_2（见随书光盘）中，玩家可以拖动鼠标来移动窗口中的飞机，并按下鼠标左键来发射子弹。

DirectInput 初始化的程序代码同样是写在 OnCreate 函数中，而在这个范例的结构中，主要是将所需的图片加载给对应的 DC，并设定鼠标与飞机图片的起始位置。下面请看程序代码的说明。

**程序代码**

```

1  LPDIRECTINPUT7 pDI;
2  LPDIRECTINPUTDEVICE7 pDMO;
3  HRESULT result;
4  DIMOUSESTATE2 MState;           //声明鼠标状态结构
5  int x,y,xlast,ylast,bx,by;
6  int xp[500];
7  canvasFrame::canvasFrame()
8  {
9  int cx,cy;
    //建立窗口的程序代码（略）
10 GetClientRect(&rect);
11 ClientToScreen(&rect);           //转换坐标

```



```

12  ::ClipCursor(&rect);           //设定鼠标移动范围
    //将图文件加载对应DC 的程序代码(略)
13  x = ((rect.right-rect.left) - 100)/2;
14  y = ((rect.bottom-rect.top) - 100)/2;
15  cx = x + 81/2;
16  cy = y + 81/2;
17  CPoint *p = new CPoint(cx,cy);
18  ClientToScreen(p);             //转换坐标
19  ::ShowCursor(false);           //取消鼠标光标
20  ::SetCursorPos(p->x,p->y);      //设定鼠标位置
21  ScreenToClient(&rect);         //转换坐标
22  )
    
```

## 程序说明

- (1) 第 4 行: 声明一个存储鼠标状态的“DIMOUSESTATE2”结构“MState”。
  - (2) 第 5 行: 声明程序中将会用到的坐标变量。
  - (3) 第 6 行: 声明一个大小为 500 的数组, 用来存储每颗子弹的 x 坐标。
  - (4) 第 10 行: 取得操作窗口的大小。
  - (5) 第 11 行: 将操作窗口坐标转换为屏幕坐标(这个动作是必须的)。
  - (6) 第 13、14 行: 设定程序开始时, 飞机所在的坐标 (x,y)。
  - (7) 第 15、16 行: 设定鼠标所在的起始坐标 (cx,cy)。
  - (8) 第 19、20 行: 分别运行“ShowCursor”与“SetCursorPos”取消光标显示以及鼠标位置。
- 程序一开始的画面如图 10-7 所示。

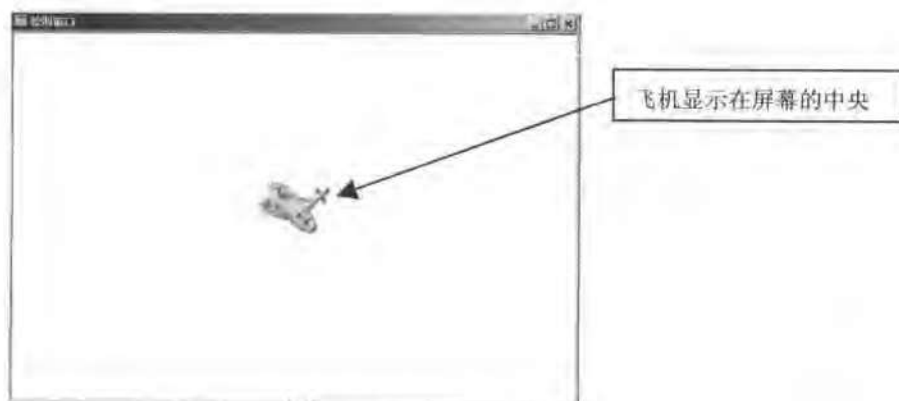


图 10-7

OnTimer 函数中, 当取得的定时器 ID 识别码为“500”时表示程序取得鼠标的输入消息, 否则表示为每一颗子弹的定时器。程序中每一颗子弹移动都需要重复贴图, 因此当程序检测到按下鼠标左键发射子弹时, 便自动建立一个新定时器来进行贴图的动作, 下面将对程序内容做详细说明。

## 程序代码

```

1  void canvasFrame::OnTimer(UINT nIDEvent)
2  {
3      CFrameWnd::OnTimer(nIDEvent);
4      CClientDC dc(this);
5      if(nIDEvent == 500)           //判断输入消息
    
```

```

6  {
7      result = pDMO->GetDeviceState(sizeof(MState),
8      (LPVOID)&MState);           //取得鼠标状态
9      if(result != DI_OK )
10         MessageBox("取得键盘状态失败!");
11      x += MState.lX;               //设定飞机图标的 x 坐标
12      y += MState.lY;               //设定飞机图标的 y 坐标
13      if(x<rect.left)               //是否已至左边界
14         x = rect.left;
15      if(x>rect.right-100)           //是否已至右边界
16         x = rect.right-100;
17      if(y<rect.top)                 //是否已至上边界
18         y = rect.top;
19      if(y>rect.bottom-74)           //是否已至下边界
20         y = rect.bottom-74;
21      dc.BitBlt(xlast,ylast,100,74,mdc,0,0,WHITENESS);
22      //覆盖上次的贴图
23      xlast = x;
24      ylast = y;
25      if(MState.rgbButtons[0] & 0x80) //判断是否按下鼠标左键
26      {
27          bx = x;                     //子弹起始的 x 坐标
28          by = y+30;                 //子弹起始的 y 坐标
29          SetTimer(by,100,NULL);      //设定定时器
30          xp[by] = bx;               //存储起始 x 坐标
31      }
32      dc.BitBlt(x,y,100,74,mdc,0,0,SRCCOPY); //贴上飞机
33  }
34  else
35  {
36      int xshut,yshut;
37      xshut = xp[nIDEvent];           //取得起始的 x 坐标
38      yshut = nIDEvent;               //取得起始的 y 坐标
39      dc.BitBlt(xshut,yshut,32,15,mdc1,0,0,SRCCOPY);
40      //贴上子弹
41      xshut-=10;
42      if(xshut<-30)
43          KillTimer(nIDEvent);        //删除定时器
44      else
45          xp[nIDEvent] = xshut;       //设定下一次子弹出现的位置
46  }
47  }

```

#### 程序说明

- (1) 第 5 行: 判断接收到的 WM\_TIMER 消息是哪个定时器发出的, 除了识别码为“500”的定时器发出的消息是再次取得鼠标的状态外, 其他都为重绘子弹的位置。
- (2) 第 11、12 行: 按照鼠标的移动量重设要贴上飞机图的位置。
- (3) 第 13~20 行: 判断飞机的位置是否已到达窗口边界, 若到达边界则设定固定的 x 或 y 点

坐标。

(4) 第 23、24 行：将其存入 `xlast` 与 `ylast` 变量中，以便下一次 `OnTimer` 函数运行时能够覆盖上一次的贴图。

(5) 第 25 行：判断鼠标左键是否被按下，若左键被按下，则设定子弹发射起点的位置 `bx` 与 `by`。

(6) 第 29 行：以 `by` 为识别码为每一个子弹建立一个定时器。

(7) 第 35~46 行：若 `OnTimer` 函数接收到定时器发出的消息，而其识别码若不是 500，则此段程序代码便完成子弹移动贴图的动作。

(8) 第 37、38 行：取得子弹发射的起点“`xshut`”与“`yshut`”。第 39 行的程序代码为绘制子弹；第 41 行的程序代码递减 `xshut`（子弹 `x` 坐标）的值。

(9) 第 42 行：判断子弹的位置是否已经到达窗口的最左边，若是，则删除代表该子弹的定时器，否则，将 `xshut` 的值存入 `xp` 数组中，以便下次取得子弹的显示位置。

## 运行结果

程序的运行结果如图 10-8 所示。

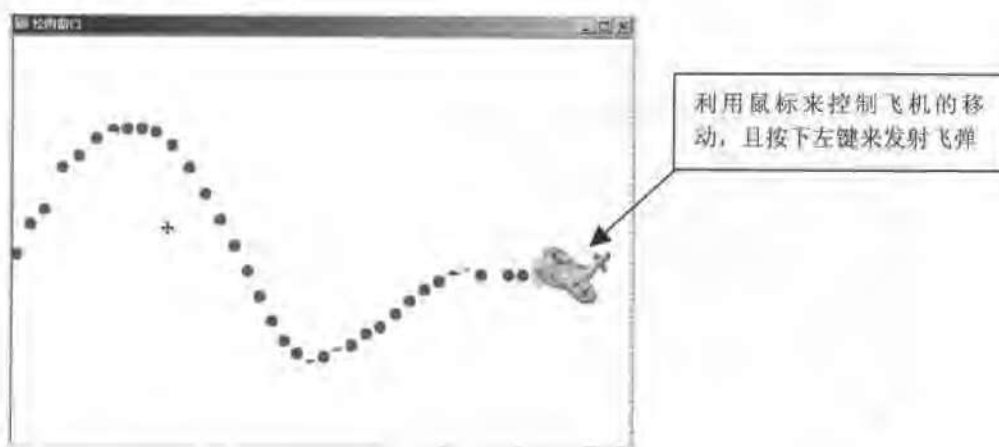


图 10-8

看过范例 `ch10_2`（见随书光盘），相信你已经学会了如何设计 `DirectInput` 程序并使用鼠标来做输入的装置。接下来一节中将进一步讨论使用 `DirectInput` 取得游戏摇杆输入消息的技巧。

## 10.3 使用摇杆功能

摇杆的种类可以说是五花八门，其结构也比键盘与鼠标复杂的多。一支摇杆上可能有许多个控制轴，或多个按钮；而一台计算机上也可能不只安装一支摇杆，因此在使用 `DirectInput` 来取得摇杆装置的输入时，便多了几个步骤。例如，检测计算机上的各种摇杆装置，以及每一支不同摇杆上的组件（控制轴与按钮）类型。下面看看如何利用 `DirectInput` 以摇杆为输入装置取得玩家所输入的控制消息。

### 10.3.1 取得摇杆装置

要以摇杆为输入装置，同样必须先建立 `DirectInput` 对象，并取得可使用的摇杆装置，代码如下所示：

## 程序代码

```

1  LPDIRECTINPUT7      pDI;                //声明 DirectInput 对象指针
2  LPDIRECTINPUTDEVICE7 pDJS;              //声明输入装置对象指针
3  HRESULT result;                        //声明 HRESULT 类型变量
4  HINSTANCE hinst = AfxGetInstanceHandle();
5  result = DirectInputCreateEx(hinst, DIRECTINPUT_VERSION,
6  IID_IDirectInput7, (LPVOID*)&pDI, NULL );
7  if(result != DI_OK)
8  MessageBox("建立 DirectInput 对象失败!");
9  result = pDI->EnumDevices(DIDEVTYPE_JOYSTICK, Joysticks,
10 NULL, DIEDFL_ATTACHEDONLY );           //列举摇杆装置
11 if(result != DI_OK)
12 MessageBox("列举摇杆失败!");
13 if(pDJS == NULL)                        //判断是否找到摇杆
14 MessageBox("无可使用的摇杆装置!");

```

## 程序说明

(1) 第 1~8 行: 声明程序要使用的对象指针, 并建立 DirectInput 对象“pDI”。其中第 2 行程序代码声明一个输入装置指针“pDJS”, 代表在计算机上找到的摇杆装置。

(2) 第 9 行: 调用“EnumDevices”方法来列举计算机上可用的摇杆, 其中输入的参数及其说明见表 10-4。

表 10-4

参 数	说 明
DIDEVTYPE_JOYSTICK	参数 1 为所要列举装置的形式, DIDEVTYPE_JOYSTICK 表示为摇杆
Joysticks	参数 2 是一个调用函数 Joysticks, 当找到新的输入装置时便会调用此函数, 会将建立摇杆装置对象的程序代码写在这个函数中
NULL	参数 3 是设定一个要传给调用函数的值, 在此设为 NULL
DIEDFL_ATTACHEDONLY	参数 4 表示所要列举装置的状态, DIEDFL_ATTACHEDONLY 表示要列举的为目前连接在计算机上的摇杆

(3) 第 13 行: 判断在列举摇杆后, 是否建立了摇杆输入装置对象“pDJS”, 若该对象为 NULL, 表示在计算机上并没有可以使用的摇杆设备。

建立摇杆输入装置对象, 在列举摇杆时必须调用函数“Joysticks”中的内容。

## 程序代码

```

1  BOOL CALLBACK Joysticks(LPCDIDEVICEINSTANCE lpddi, LPVOID pvRef)
2  {
3  result = pDI->CreateDeviceEx( pddiInstance->guidInstance,
4  IID_IDirectInputDevice7, (VOID**)&pDJS, NULL );
5  //建立输入装置对象
6  if(result != DI_OK)
7  return DIENUM_CONTINUE;
8  else
9  return DIENUM_STOP;
10 }

```

## 程序说明

(1) 第 1 行：此函数输入的参数“pddiInstance”指向输入装置的实体，“pContext”是传给此函数的值（在前面设为 NULL）。

(2) 第 3、4 行：调用“CreateDeviceEx”方法来建立输入装置对象，其中输入的参数及其说明见表 10-5。

表 10-5

参 数	说 明
lpddi->guidInstance	参数 1 输入要建立输入装置的识别码，pddiInstance->guidInstance 取得装置的识别码
IID_IDirectInputDevice7	参数 2 输入使用接口的识别码，在此设为 IID_IDirectInputDevice7
(VOID**) &pJS	参数 3 须输入建立对象的指针 pJS，并转换为 LPVOID 类型
NULL	参数 4 须设为 NULL

(3) 此函数返回“DIENUM\_CONTINUE”值表示继续搜寻其他可用的装置，或者“DIENUM\_STOP”表示停止搜寻其他的装置。

## 10.3.2 摇杆组件的列举方法

到目前为止，如果在计算机上插上了摇杆，那么 10.3.1 节中所讨论的程序代码应该已经可以为这个摇杆建立一个输入装置对象。

不过，还不知道找到的摇杆装置上有多少按钮和控制轴，所以接下来必须要做的就是取得这个摇杆的特性，并列举其可使用的组件。如此，程序才能完全掌握通过摇杆输入的各项消息。下面先来了解一下摇杆上可能出现的组件类型：

- (1) 轴型：摇杆上可能出现的轴有“x、y、z 方向轴”，或者“x、y、z 的旋转轴”，以及“滑轴”。
- (2) 按钮：摇杆上任何位置上的独立按钮。
- (3) 准星帽：同样控制方向，但具有较高的灵敏度。

下面以图 10-9 来说明以上三种摇杆组件的概念。

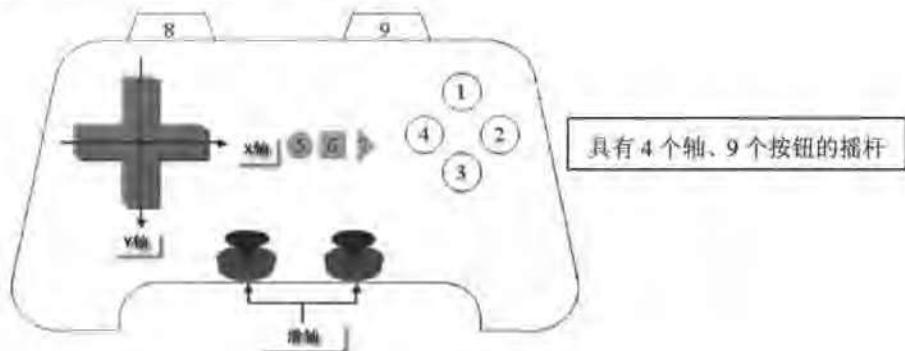


图 10-9

也可以开启控制台中“Gaming Options”项目，选择使用的摇杆来查看其他具有怎样的组件与功能，如图 10-10 所示。

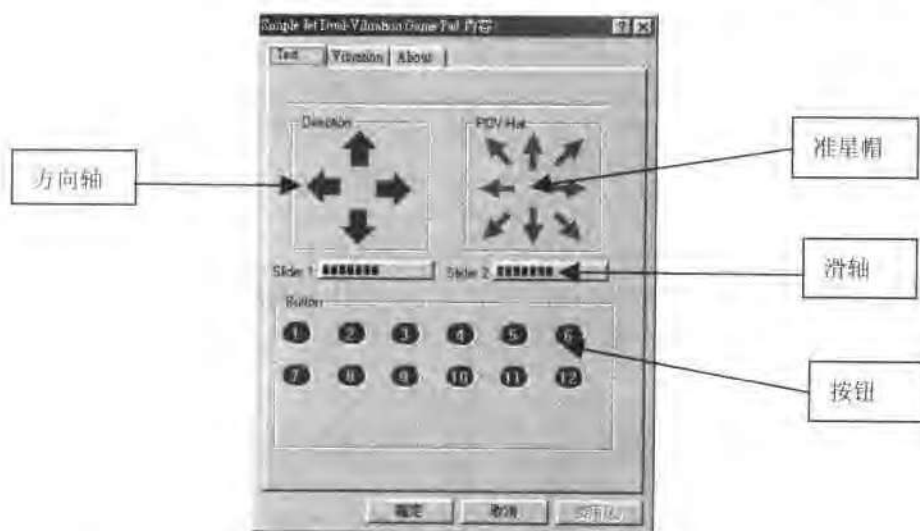


图 10-10

在认识了摇杆上组件的类型后，下面来看看取得摇杆特性和列举摇杆组件的程序代码。

#### 程序代码

```

1  DIDEVCAPS JSCap;           //声明摇杆特性结构
2  result = pDJS->SetDataFormat(&c_dfDIJoystick);
3  if(result != DI_OK)
4  MessageBox("设定资料格式失败!");
5  result = pDJS->SetCooperativeLevel(m_hwnd,
6  DISCL_EXCLUSIVE|DISCL_BACKGROUND);
7  if(result != DI_OK)
8  MessageBox("设定协调层级失败!");
9  JSCap.dwSize = sizeof(JSCap);
10 result = pDJS->GetCapabilities(&JSCap);
11 if(result != DI_OK)
12 MessageBox("取得摇杆特性失败!");
13 result = pDJS->EnumObjects(Objects, (VOID*)m_hwnd, DIDFT_AXIS);
14 if(result != DI_OK)
15 MessageBox("列举摇杆组件失败!");
16 result = pDJS->Acquire();
17 if(result != DI_OK)
18 MessageBox("取得摇杆装置失败!");

```

#### 程序说明

- (1) 第 1 行：声明一个“DIDEVCAPS”类型的结构“JSCap”。
- (2) 第 10 行：调用“GetCapabilities”方法将该摇杆的特性存入结构中，这个结构比较重要的参数见表 10-6。
- (3) 第 2~8 行：设定资料格式与协调层级，其中设定资料格式时必须输入参数“&c\_dfDIJoystick”。

表 10-6

参 数	说 明
dwSize	结构的大小，必须在调用 GetCapabilities 方法之前设定此资料成员
dwAxes	摇杆上轴的数目
dwButtons	摇杆上按钮的数目
dwPOVs	摇杆上准星帽的数目

(4) 第 13 行：调用“EnumObjects”方法来列举摇杆上的组件，这里只列举“轴”并输入回调函数来设定轴的范围与无效区，输入参数及其说明见表 10-7。

表 10-7

参 数	说 明
Objects	参数 1 是一个回调函数 Objects，当在摇杆上找到新的组件时便会调用此函数，设定摇杆上各个轴的特性的程序代码便定义在此函数中
(VOID*)m_hWnd	参数 2 则是输入窗口的 handle，并转换为 VOID 类型
DIDFT_AXIS	参数 3 设定为 DIDFT_AXIS，表示在列举的组件为轴

接下来再列举出所有摇杆上的轴，并设定 x 与 y 轴的最大最小范围来调整移动时的灵敏度。下面是程序内容。

### 程序代码

```

1  BOOL CALLBACK Objects(LPCDIDEVICEOBJECTINSTANCE
2  lpddoi, LPVOID pvRef)
3  {
4      DIPROPRANGE dr;
5      dr.diph.dwSize = sizeof(DIPROPRANGE);
6      dr.diph.dwHeaderSize = sizeof(DIPROPHEADER);
7      dr.diph.dwHow = DIPH_BYOFFSET;
8      dr.diph.dwObj = lpddoi->dwOfs; //取得轴的类型
9      switch( lpddoi->dwOfs )
10     {
11         case DIJOFS_X:
12             dr.lMin = -100;
13             dr.lMax = 100;
14             result = pDJS->SetProperty(DIPROP_RANGE,&dr.diph);
15             if(result != DI_OK)
16                 MessageBox(NULL, "设定轴范围失败!", "错误消息", MB_OK);
17             break;
18         case DIJOFS_Y:
19             dr.lMin = -50;
20             dr.lMax = 50;
21             result = pDJS->SetProperty(DIPROP_RANGE,&dr.diph);
22             if(result != DI_OK)
23                 MessageBox(NULL, "设定轴范围失败!", "错误消息", MB_OK);
24             break;
25         case DIJOFS_Z:
26             //设定 z 方向轴范围的程序代码(略)

```

```

26         break;
27     case DIJOFS_RX:
28         //设定 x 旋转轴范围的程序代码 (略)
29         break;
30     case DIJOFS_RY:
31         //设定 y 旋转轴范围的程序代码 (略)
32         break;
33     case DIJOFS_RZ:
34         //设定 z 旋转轴范围的程序代码 (略)
35         break;
36     case DIJOFS_SLIDER(0):
37         //设定滑轴 1 范围的程序代码 (略)
38         break;
39     case DIJOFS_SLIDER(1):
40         //设定滑轴 2 范围的程序代码 (略)
41         break;
42 }
43 return DIENUM_CONTINUE; //继续搜寻下一个轴
44 }

```

#### 程序说明

- (1) 第 1、2 行: 输入回叫函数的参数“lpddoi”为列举组件的指针;“pvRef”为指定给这个回叫函数的值。
- (2) 第 4 行: 定义一个“DIPROPRANGE”类型的结构“dr”用来描述轴的范围。
- (3) 第 5~7 行: 设定结构的大小与描述组件的方法。
- (4) 第 8 行: 用“lpddoi->dwOfs”取得轴的类型, 各种摇杆轴的类型在 DirectInput 中的默认值见表 10-8。

表 10-8

默 认 值	说 明
DIJOFS_X	X 方向的轴
DIJOFS_Y	Y 方向的轴
DIJOFS_Z	Z 方向的轴
DIJOFS_RX	X 方向的转动轴
DIJOFS_RY	Y 方向的转动轴
DIJOFS_RZ	Z 方向的转动轴
DIJOFS_SLIDER(0)	滑轴 1
DIJOFS_SLIDER(1)	滑轴 2

(5) 第 38 行: 返回值为“DIENUM\_CONTINUE”, 因此这个回叫函数会一一列举出摇杆上所有的轴。第 9 行程序代码利用 switch 判断式判断列举轴的类型。

(6) 以 11~17 行程序代码设定 X 方向轴的范围为例, 第 12 行的程序代码设定最小值为“-100”, 即按下 X 轴的左边时, 最大移动范围为 100; 第 13 行的程序代码设定最大值为“100”, 即按下 X 轴的右边时, 最大移动范围为 100。

(7) 第 14 行: 调用“SetProperty”函数设定 X 方向轴的范围。

(8) 第 18~24 行: 按照相同的方式设定 Y 轴的范围为-50~50。



(9) 第 25~36 行: 若程序中还必须用到其他的轴, 则根据各个轴的默认值在程序代码中定义它的特性与范围。

完成了以上的步骤之后, 程序便已经设定好了所要使用轴的范围, 并且可以取得摇杆输入的消息了。

## 10.3.3 摇杆输入的取得

要取得摇杆的输入消息, 同样必须不断地在每一时刻取得摇杆上的最新输入状态。此外, 有些摇杆装置还必须在取得其输入状态之前做“检测 (Poll)”的操作, 才能得到新的输入信息, 下面我们就对如何检测摇杆状态以及取得输入资料的方法进行说明。

### 1. 检测状态

检测摇杆状态的这个步骤并非必要, 但是为了让程序能够适用于某些必须要进行检测才能取得输入资料的摇杆, 通常也会在程序中加入检测的动作。检测摇杆状态的程序代码很简单, 如下面的这行叙述:

```
result = pDJS->Poll();
```

在这行叙述中, 摇杆对象“pDJS”会调用“Poll”方法进行检测, 并返回值给“result”。若检测成功, 则 result 等于“DI\_OK”; 若该摇杆不需进行检测的操作则会返回“DI\_NOEFFECT”。

### 2. 取得状态

对于摇杆, 有一个“DIJOYSTATE”的结构存储其输入状态, 该结构内容如下:

```
typedef struct DIJOYSTATE {
    LONG    lX;           //X 方向轴的移动量
    LONG    lY;           //Y 方向轴的移动量
    LONG    lZ;           //Z 方向轴的移动量
    LONG    lRx;          //X 旋转轴的转动量
    LONG    lRy;          //Y 旋转轴的转动量
    LONG    lRz;          //Z 旋转轴的转动量
    LONG    rgfSlider[2]; //代表滑轴是否被按下
    DWORD   rgdwPOV[4];  //代表准星帽是否被按下
    BYTE    rgbButtons[32]; //代表按钮是否被按下
} DIJOYSTATE, *LPDIJOYSTATE;
```

下面的程序代码可以取得目前摇杆的输入状态, 并存入“DIJOYSTATE”类型的“JState”结构中。

#### 程序代码

```
1  DIJOYSTATE JState;           //声明摇杆状态结构
2  result = pDJS->GetDeviceState( sizeof(JState), &JState );
3  if (result != DI_OK)
4  MessageBox("取得摇杆状态失败!");
```

#### 程序说明

(1) 第 1 行: 声明一个 JState 结构。

(2) 第 2 行: 调用“GetDeviceState”将目前摇杆的状态存入 JState 中, 再对照 DIJOYSTATE 结构的资料成员内容, 就可以知道目前摇杆的状态了。

接下来看范例 ch10\_3 (见随书光盘), 这个范例是以摇杆当作控制飞机移动以及子弹发射的输入装置。这个范例的架构和流程和 ch10\_2 相同, 以摇杆为输入装置的初始化方法也在前面的内容中讨论过, 所以直接来看 OnTimer 函数中以摇杆控制飞机移动与子弹发射的程序代码内容:

#### 程序代码

```

1 void canvasFrame::OnTimer(UINT nIDEvent)
2 {
3     CFrameWnd::OnTimer(nIDEvent);
4     CClientDC dc(this);
5     result = pDJS->Poll();           //检测摇杆
6     if(result != DI_OK && result != DI_NOEFFECT)
7         MessageBox("检测摇杆失败!");
8     result = pDJS->GetDeviceState( sizeof(JState), &JState );
9     if (result != DI_OK)
10        MessageBox("取得摇杆状态失败!");
11    x += JState.lX;                    //设定飞机图标的 x 坐标
12    y += JState.lY;                    //设定飞机图标的 y 坐标
13    if(x<rect.left)                    //是否已至左边界
14        x = rect.left;
15    if(x>rect.right-100)                //是否已至右边界
16        x = rect.right-100;
17    if(y<rect.top)                     //是否已至上边界
18        y = rect.top;
19    if(y>rect.bottom-74)                //是否已至下边界
20        y = rect.bottom-74;
21    dc.BitBlt(xlast,ylast,100,74,mdc,0,0,WHITENESS);
22    //覆盖上次的贴图
23    xlast = x;
24    ylast = y;
25    if(JState.rgbButtons[0] & 0x80)    //判断是否按下发射子弹按钮
26    {
27        for(i=0;i<1000;i++)
28        {
29            if(b[i].exist == false)
30            {
31                b[i].x = x;
32                b[i].y = y+30;
33                b[i].exist = true;
34                bcount++;
35                break;
36            }
37        }
38    }
39    if(JState.rgbButtons[1] & 0x80)    //是否按下结束按钮
40        PostMessage(WM_CLOSE);        //传送 WM_CLOSE 消息
41    if(bcount != 0)

```

```

42  for(i=0;i<1000;i++)
43  {
44      dc.BitBlt(b[i].x,b[i].y,32,15,mdcl,0,0,SRCCOPY);
45      b[i].x -=17;
46      if(b[i].x<-30)
47      {
48          b[i].exist = false;
49          bcount--;
50      }
51  }
52  dc.BitBlt(x,y,100,74,mdc,0,0,SRCCOPY);    //贴上飞机
53  }

```

## 程序说明

- (1) 第 5~8 行：先对摇杆进行检测，把取得目前的状态存入“JState”中。
- (2) 第 11、12 行：取得摇杆在这两个方向上的移动量，并重新计算飞机贴图的新坐标。
- (3) 第 25 行：判断玩家是否按下了摇杆上的发射子弹按钮，若是，则新增一颗子弹到数组中。
- (4) 第 39 行：判断玩家是否按下了摇杆上的结束按钮，若是，则发出“WM\_CLOSE”消息结束程序。

## 10.3.4 设定无效范围

“无效范围”的意义也就是说，当玩家使用摇杆时，在某一轴上的移动量若是在无效范围内，则程序将不予理会，并将其输入的消息视为无效。

设定无效范围的用意通常是在为了防止使用者不小心误触摇杆，或者是预防摇杆本身的偏移。以上一个范例而言，就使用的摇杆为例，当程序开始时，飞机便会自动移动，这是因为摇杆本身并没有完全归零所致。

这时可以设定无效范围，让程序忽略掉一些轻微的移动消息。设定无效范围与设定轴的最大最小范围其实是很相似的，所以必须建立一个“DIPROPDWORD”类型的结构，定义其资料成员的内容，并且调用“SetProperty”方法来设定无效范围。其表现方式如下面程序代码所示。

## 程序代码

```

1  DIPROPDWORD du;
2  du.diph.dwSize = sizeof(DIPROPDWORD);
3  du.diph.dwHeaderSize = sizeof(DIPROPHEADER);
4  du.diph.dwObj = DIJOFS_X;
5  du.diph.dwHow = DIPH_BYOFFSET;
6  du.dwData = 2000;
7  result = pDJS->SetProperty(DIPROP_DEADZONE,&du.diph);
8  if(result != DI_OK)
9      MessageBox(NULL,"设定无效范围失败!", "错误消息", MB_OK);

```

## 程序说明

- (1) 第 1 行：定义一个“DIPROPDWORD”类型的结构“du”。
- (2) 第 4 行：设定“dwObj”资料成员为“DIJOFS\_X”，表示要设定无效范围的组件为 X 轴。
- (3) 第 6 行：设定“dwData”资料成员为“2000”，表示无效范围为 X 轴范围的 20%。可设定

的值最大为“10000”，表示无效范围为 100%，即该轴上的移动都无效。

(4) 第 7 行：调用“SetProperty”方法，输入参数 1 为“DIPROP\_DEADZONE”表示设定无效范围。

若要为范例 ch10\_3 中的 X 轴与 Y 轴设定无效范围，在同叫函数 Objects 中的部分程序代码内容如下：

#### 程序代码

```
//定义 DIPROPRANGE 结构的程序代码 (略)
1  DIPROPDWORD du;
2  du.diph.dwSize = sizeof(DIPROPDWORD);
3  du.diph.dwHeaderSize = sizeof(DIPROPHEADER);
4  du.diph.dwHow = DIPH_BYOFFSET;
5  switch( lpddoi->dwOfs )
6  {
7  case DIJOFS_X:
8      dr.lMin = -100;
9      dr.lMax = 100;
10     result = pDJS->SetProperty(DIPROP_RANGE,&dr.diph);
11     if(result != DI_OK)
12         MessageBox(NULL,"设定轴范围失败!","错误消息",MB_OK);
13     du.diph.dwObj = DIJOFS_X;
14     du.dwData = 2000;
15     result = pDJS->SetProperty(DIPROP_DEADZONE,&du.diph);
16     if(result != DI_OK)
17         MessageBox(NULL,"设定无效范围失败!","错误消息",MB_OK);
18     break;
19 case DIJOFS_Y:
20     dr.lMin = -50;
21     dr.lMax = 50;
22     result = pDJS->SetProperty(DIPROP_RANGE,&dr.diph);
23     if(result != DI_OK)
24         MessageBox(NULL,"设定轴范围失败!","错误消息",MB_OK);
25     du.diph.dwObj = DIJOFS_Y;
26     du.dwData = 1000;
27     result = pDJS->SetProperty(DIPROP_DEADZONE ,&du.diph);
28     if(result != DI_OK)
29         MessageBox(NULL,"设定无效范围失败!","错误消息",MB_OK);
30     break;
31 //设定其他轴的程序代码 (略)
32 }
```

#### 程序说明

(1) 第 1~4 行：定义“du”结构并设定共享的资料成员。

(2) 第 13~15 行：设定 X 方向轴的无效范围为 20% (du.dwData=2000)。

(3) 第 25~27 行：设定 Y 方向轴的无效范围为 10% (du.dwData=1000)。

完成了以上的设定，在范例 10\_3 中若 X 方向轴的波动小于 20%，Y 方向轴的波动小于 10%，则飞机的位置将不会有任何的移动。

## 课后重点整理

- ### 课后习题

- 表 10-9

(5) 设定无效范围的用意通常是什么?

- 292

## 第 11 章 威力强大的 DirectPlay 和 DirectShow

亮丽的画面或动人的音乐可以轻易地吸引大众的目光，如果能再击败一个活生生的对手就更有成就感了，因为再怎么精心设计的人工智能都比不上人类的机智。电视游戏发展初期，早已提供“多人共玩”的环境，Windows 系统上虽然可以使用 DirectInput 做到单机多人的游戏，但还是得事先找些志同道合的朋友，约好时间、地点才可以进行厮杀。

任何时间、空间，都可以找到共同参与的伙伴一起进行游戏，就是网络游戏的基本概念。随着时代的进步，网络逐渐成为计算机的标准配备，也改变了人们玩计算机游戏的方式，“网络化”已不知不觉成为游戏的“必要配备”。

### 11.1 DirectPlay 初体验

在介绍完 DirectX 的绘图系统 (Direct Graphics)、播放音乐 (DirectSound) 及使用者操作接口 (DirectInput) 之后，接下来还要介绍另一种较为特殊且当下最为流行的网络功能——DirectPlay。

Windows 提供名为“Winsock”的网络套件，与其他 API 一样简单易学，不过许多基本工作（如联机建立、玩家管理、封包传递）还得靠软件工程师自己动手 DIY。对于简单的小游戏它使用起来相当方便，但如果要达到实用的阶段，还是得花更多时间进行深入研究。而 DirectPlay 则包含了网络游戏的基本架构，让软件工程师可以快速上手并轻松使用。

#### 11.1.1 DirectPlay 的使用时机

DirectPlay 在 DirectX 的系列中的主要功能是提供控制网络上数据传输的函数特性，以方便开发者的使用。

在开发网络型的游戏时，可以利用 DirectPlay 提供的网络函数库进行开发；或者不使用 DirectPlay 的函数库，直接使用 Win32 本身所提供的 TCP/IP 的 API 函数。

在使用 DirectPlay 来开发网络架构的时候，DirectPlay 可以协助分析出几种不同的通讯协议之间的差异性。简单地说，如果使用 DirectPlay 的话，可以编写一套支持 IPX、TCP/IP、串联、调制解调器通讯，以及任何其他不同网络通讯协议的程序代码。如果没有使用 DirectPlay，就必须自行开发这些不同且繁杂琐碎的通讯协议程序代码。相比之下，DirectPlay 的确可以省下不少的开发时间。

其实严格讲起来，使用 DirectPlay 也不完全是一件好事，因为它局限在 Windows 的平台下。不过到目前为止，使用 Windows 平台来开发游戏的公司在市面上还是占绝对大多数的。

#### 11.1.2 DirectPlay 的网络拓扑

通常 DirectPlay 可以支持两种网络拓扑的方式，即：

- (1) 点对点模式 (Peer-to-Peer)

### (2) 服务器/客户端模式 (Client/Server)

如果对于网络拓扑的原理不太了解的话, 请看下面的介绍。

#### 1. 点对点模式

在点对点的网络拓扑中, 并没有中央服务器的设备, 终端计算机会直接与其他计算机做网络联机的操作, 而它必须要自己设法接收或传送消息给其他的计算机以便了解到它做了什么事。例如, 在点对点的网络上, 某一计算机要传送消息给其他计算机, 这台计算机就要负起与其他计算机联机的任务。如图 11-1 所示。

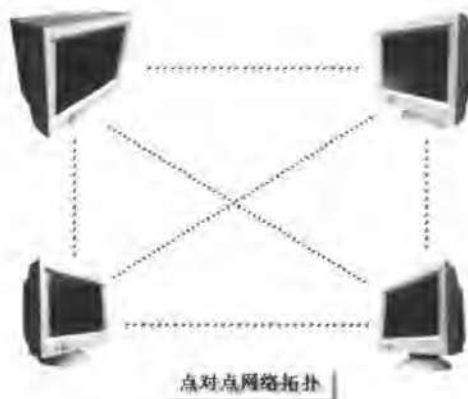


图 11-1

#### 2. 服务器/客户端模式

终端计算机与其他的计算机不会直接进行联机的操作, 在网络上所有计算机的消息都必须经过服务器来传送至其他计算机上。举个例子来说, 如果网络游戏中的某一个玩家的主角在游戏中向前走了两步, 此终端计算机便会向服务器传送走了两步的消息, 然后再由服务器将此消息传送至网络上的每一台终端机上。如此一来, 便可以大大地减少终端机节点的网络负担。此过程如图 11-2 所示。



图 11-2

在 DirectPlay 的应用上, 可以自行决定使用哪一种方式建构网络。例如, 就点对点而言, 可以使用 IDirectPlay8Peer 的接口来进行联机, 就服务器/客户端而言, 在服务器上可以使用

DirectPlay8Server 接口来构成, 在客户端上, 可以使用 DirectPlay8Client 接口来构成。

### 11.1.3 网络联机游戏的构成

在利用 DirectPlay 的函数库来开发游戏之前, 必须要规划出一系列的使用者基本程序。这个基本程序分类如下所示:

- (1) 建立 DirectPlay 接口。
- (2) 列举 DirectPlay 服务提供者。
- (3) 建立游戏或加入游戏。
- (4) 列出游戏与玩家的名称。
- (5) 列举所有可进入的游戏。
- (6) 处理消息。
- (7) 传送与接收消息。
- (8) 结束联机。

在网络联机的游戏中, 不外乎上述的八项原则, 接下来介绍这些原则所代表的功能与意义。

#### (1) 建立 DirectPlay 接口

当玩家进入到游戏的标题或主页的时候, 可以在这里放置一个多人联机的选项, 并且在此时建立起 DirectPlay 的接口。

#### (2) 列举 DirectPlay 服务提供者

当玩家点击多人联机的选项后, 就必须先列举出 DirectPlay 提供的通讯服务协议, 例如 IPX 或 TPC/IP 等。

#### (3) 建立游戏或加入游戏

当玩家选择通讯协议后, 接下来要让使用者选择加入或自己建立一个新的游戏。例如, 玩家选择了服务器/客户端的模式之后, 就必须按照玩家所选择的模式来建构一个服务器或客户端的 DirectPlay 接口。

#### (4) 列出游戏与玩家的名称

如果玩家所选择的是建立一个新游戏, 那么就必须让玩家可以列出新游戏的名称, 以及其他的玩家名称。在点对点的拓扑中, 可以使用 SetPeerInfo 来设定信息; 在服务器/客户端的拓扑中, 可以使用 SetClientInfo 来设定信息。

#### (5) 列举所有可进入的游戏

如果玩家所选择的是加入游戏, 就必须根据玩者所选择的接口方式(点对点或服务器/客户拓扑)中的 EnumHosts 方法来列举所有可用的游戏。简单的说, 玩家可以在这里选择他想要进入的游戏或机器, 最后再利用 Connect 联机。

#### (6) 处理消息

当玩家建立或连接游戏的时候, 玩家可以与其他的玩家聊天、等待其他人的联机或在游戏开始前进行游戏的设定。这个时候, 游戏必须送出或接收消息, 并与其他计算机取得联系, 以通知网络上的每一台计算机其设定上的更改, 而在 DirectPlay 函数中, 可以去处理“DPN\_MSGID\_INDICATE”及“DPN\_MSGID\_CREATERPLAYER”的消息。

#### (7) 传送与接收消息

在玩家与其他的计算机进行游戏时, 也可以利用 Send 的方法来传送消息(客户端), 或以 SendTo





的方法来传送消息（服务器端或点对点拓扑）。

在消息传送期间，可以使用“DPN\_MSGID\_RECEIVE”的消息来接收从其他地方传来的资料。

#### (8) 结束联机

在玩家与其他计算机打算结束联机的时候，可以使用 Close 的方法取消联机。

接下来还要针对上述几项原则深入讨论其 DirectPlay 的使用方法。

### 11.1.4 DirectPlay 的组成模式

在 DirectPlay 建构之前，先介绍一种网络消息调用的方法，就是利用 Windows 的响应函数与游戏程序沟通，这个响应函数的功能就是负责接收 DirectPlay 的消息，它的作用有点像之前介绍过的 Windows 消息。

其函数语法如下所示：

```
1 HRESULT WINAPI DirectPlayMesHandle
2 (
3     PVOID pUserContext,
4     DWORD dwMessage,
5     PVOID pBuffer
6 );
```

(1) 第 3 行：其参数用来存放应用程序定义的结构指针。

(2) 第 4 行：消息资料。此参数用来接收 DirectPlay 传来的消息，它可以是 Windows 的“WM\_”开头的消息。一般来说，用它接收“DPN\_MSGID\_”开头的 DirectPlay 消息。

(3) 第 5 行：此参数可以依据 dwMessage 的值来指定不同事件的指针，不过并非所有的消息都有结构，所以一般来说此参数始终都为 NULL。

了解了消息的接收方式后，接下来将利用之前讲述过的网络联机构成流程来深入讨论 DirectPlay 的函数使用方法。

#### 1. 选择服务提供者

一般说来，大多数的游戏会在游戏的第一个画面中让玩家选择一个适合自己的服务通讯协议（如 IPX、TCP/IP、串联、调制解调器等等），所以必须将它们一一列举出来，以便玩家可以选择它们。

当要列举这些通讯服务协议时，可以调用 EnumServiceProviders。EnumServiceProviders 与其他 Enum 函数的使用方法有所不同，不同之处在于它不用响应函数来调用，而且只有一个数组来容纳计算机中的信息。如下所示：

```
1 HRESULT EnumServiceProviders
2 (
3     const GUID *const pguidServiceProvider,
4     const GUID *const pguidApplication,
5     DPN_SERVICE_PROVIDER_INFO *const pSPInfoBuffer,
6     PDWORD const pcbEnumData,
7     PDWORD const pcReturned,
8     const DWORD dwFlags
```

9 );

(1) 第 3 行: 指定是否要列举所有的通讯服务协议或特定通讯服务协议的子设备。如果要列举所有的通讯服务协议时, 将其参数设定为 NULL; 如果是列举特定通讯协议的子设备时, 只要将其 GUID 存放到此参数就可以了。

(2) 第 4 行: 其参数用法与第 3 行的参数非常类似, 也可以在这里指定是否要列举所有的服务通讯协议或特定通讯服务协议的子设备。

(3) 第 5 行: 指定一个 DPN\_SERVICE\_PROVIDER\_INFO 的结构数组指针。DirectPlay 会在其参数中填入可用的通讯服务协议信息。

(4) 第 6 行: 在其参数中可以填入一个数字, 以指定数组大小来正确地取得所有的信息。

(5) 第 7 行: 调用完成时, 此参数会返回一个成功的结构数量。

(6) 第 8 行: 在这里, 其参数只能输入一个“DPNENUMSERVICEPROVIDERS\_ALL”的标识符, 代表将所有可用的通讯服务协议列举出来。

如果数组太小, 无法容纳整个通讯服务协议清单时, 此方法会返回一个“DPNERR\_BUFFERTOOSMALL”的消息, 由存放“pcbEnumData”的值, 则可以得知需要多大的数组容量。

这个函数有一种奇怪的现象, 就是在第一次调用此方法, 而且不知道数组大小的情况下, 会返回一个错误的消息, 并且告诉我们需要定义多大的数组容量。利用第一次返回来的数组大小来定义消息的数组, 并且利用新的数组大小再调用一次此函数, 这样便可以轻易得到通讯服务协议的清单了。

## 2. 建立新的联机游戏

假设现在已经建立起一个 DirectPlay8Server 或 DirectPlayPeer 的环环接口, 接下来就开始建立一个新的联机机制, 方法如下:

首先借助 SetServerInfo 的方法 (已有 DirectPlayServer 接口) 或 SetPeerInfo 的方法 (已有 DirectPlayPeer 接口) 来设定玩家的名称或其他信息。

在建立起玩家的信息之后, 第二个步骤便是利用 Host 的方法来建立一个新联机。其 Host 的方法如下所示:

```
1 HRESULT Host
2 (
3     const DPN_APPLICATION_DESC *const pdnAppDesc,
4     IDirectPlay8Address **const prgpDeviceInfo,
5     const DWORD cDeviceInfo,
6     const DPN_SECURITY_DESC *const pdpSecurity,
7     const DPN_SECURITY_CREDENTIALS *const pdpCredentials,
8     VOID *const pvPlayerContext,
9     const DWORD dwFlags
10 );
```

(1) 第 3 行: 指定一个 DPN\_APPLICATION 结构指针, 其参数可输入建立一个新联机时附加的参数, 例如游戏的名称、最大容纳人数、密码等等。其参数中的资料大多数来自于玩家的输入。

(2) 第 4 行: 其参数用来指定通讯服务协议的 IDirectPlay8Address 对象的数组。

(3) 第 5 行: 其参数用来指定 prgpDeviceInfo 中地址的数目。

(4) 第 6、7 行: 两个参数由 DirectPlay 使用, 将其设为 NULL。

(5) 第 8 行: 指定建立者的背景选择参数, 一般而言, 将其设定成 NULL。

(6) 第 9 行: 与上一个方法一样, 其参数也只能输入一个标识符, 其标识符为 “DPNHOST\_OKTOQUERYFORADDRESSING”, 目的是弹出一个窗口告诉使用者允许它们设定网络联机的内容。

## 3. 联机到游戏中

联机到游戏的建构方式比建立一个新联机复杂, 因为必须先列举和显示所有有效的游戏, 如此玩家才能由此联机到游戏中。

要列举出所有有效的游戏, 可以调用 IDirectPlayPeer 或 IDirectPlayClient 的 EnumHosts 的方法来完成。其语法如下所示:

```
1  HRESULT EnumHosts
2  (
3  PDPN_APPLICATION_DESC const pApplicationDesc,
4  IDirectPlay8Address *const pdpaddrHost,
5  IDirectPlay8Address *const pdpaddrDeviceInfo,
6  PVOID const pvUserEnumData,
7  const DWORD dwUserEnumDataSize,
8  const DWORD dwEnumCount,
9  const DWORD dwRetryInterval,
10 const DWORD dwTimeOut,
11 PVOID const pvUserContext,
12 HANDLE *const pAsyncHandle,
13 const DWORD dwFlags
14 );
```

(1) 第 3 行: 此参数指定一个 PDPN\_APPLICATION\_DESC 的结构指针, 用来限制对建立者的搜寻。

(2) 第 4 行: 此参数用来指定建立该应用程序的计算机地址, 如果将其设定为 NULL, DirectPlay 则根据 pdpaddrDeviceInfo 的参数来建立计算机的位置。

(3) 第 5 行: 此参数用来指用列举时所使用的 IDirectplay8Address 的对象指针。

(4) 第 6 行: 此参数用来指定由网络所传送的建立者资料区块指针。

(5) 第 7 行: 其参数用来存放第 6 行参数所包含资料的数量。

(6) 第 8 行: 此参数送出列举资料的次数, 一般设为 0, 让 DirectPlay 使用一个合理的默认值。

(7) 第 9 行: 此参数用来指定送出列举资料时的间隔为多少毫秒。

(8) 第 10 行: 此参数用来指定 DirectPlay 在最后一次送出资料时等待多久。

(9) 第 11 行: 在与建立者计算机取得列举后, 此参数会响应存放建立者的所有消息。

(10) 第 12 行: 此参数返回一个代表操作的 DPNHANDLE 值。

(11) 第 13 行: 此参数中可以包含三个标识符值, 具体情况见表 11-1。

表 11-1

标识符值	说 明
DPNENUMHOSTS_SYNC	其代表的意义为如果在未得到建立者计算机的响应之前, 此项目则不会被列举出来
DPNEUMHOSTS_OKTOQUERYFORADDRESSING	此标识符会显示一个对话框来询问玩家关于如何连接的信息
DPENUMHOSTS_NOBROADCASTFALLBACK	如果您使用了一个广播的服务通讯协议, 此标识符则是用来命令 DirectPlay 不得使用广播

在上述三个标识符中,通常使用 DPNEUMHOSTS\_OKTOQUERYFORADDRESSING 标识符。

当玩家选择其中一个游戏后,可将它联机到游戏主机上。在连接之前,必须使用 SetClientInfo 或 SetPeerInfo 设定玩家的名称与信息,最后再调用 Connect 方法让玩家连接到游戏中。

#### 4. 接收与传送消息

当玩家连接到游戏之后,可以调用 SendTo 的方法(服务器端或在点对点的拓扑中)、或以 Send 的方法(客户端)将资料送给其他计算机。

SendTo 与 Send 的用法大致相同,惟一不同的是 SendTo 具有一个额外的命令可以将资料传送到特定的主机,而 Send 命令只能将资料传送到服务器端上,如图 11-3 所示。

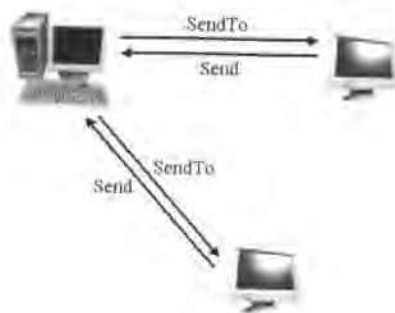


图 11-3

其中 SendTo 的参数用法如下:

```

1 HRESULT SendTo(
2     const DPNID dpnid,
3     const DPN_BUFFER_DESC *const pBufferDesc,
4     const DWORD cBufferDesc,
5     const DWORD dwTimeout,
6     void *const pvAsyncContext,
7     DPNHANDLE *const phAsyncHandle,
8     const DWORD dwFlags
9 );
  
```

(1) 第 2 行: 此参数为欲传送消息的玩家或群组 ID。

(2) 第 3 行: 指定一个 DPN\_BUFFER\_DESC 的结构指针, 此结构为描述传送的资料。

(3) 第 4 行: 指定 pBufferDesc 结构的数量, 不过在此版的 DirectPlay 架构内, 它只能传出一个结构, 所以只能将此参数设定成 1。

(4) 第 5 行: 送出消息时必须等待的毫秒数, 如果 DirectPlay 无法在这段时间里送出资料的话, 那么资料则不会被送出了。

(5) 第 6 行: 指定一个玩家背景值的指针。在 DirectPlay 传送完毕后, 可以从此参数的 ID 值了解到资料已传送完毕了。

(6) 第 7 行: 当此参数返回时, 可以利用它指向相对应的的有效 Handle 值, 以便让这个控件可以跳出传送的操作。

(7) 第 8 行: 此参数可以通过设定几个标识符来决定 DirectPlay 所要传送消息的方式, 一般来说, 最常用的有两个标识符: 一个是 "DPNSEND\_GUARANTEED", 这个标识符值能够确定消息

可传送到目的端；另一个是“DPNSEND\_PRIORITY\_HIGH”，它可以将一个消息设定成最高优先。

在游戏中有消息到达时，DirectPlay 消息控制响应函数会收到一个“DPN\_MSGID\_RECEIVE”消息，它会取得相对应的 DPNMSG\_RECEIVE 结构，其结构如下所示：

```
1  typedef struct
2  {
3      DWORD          dwSize;
4      DPNID          dpnidSender;
5      PVOID          pvPlayerContext;
6      PBYTE          pReceiveData;
7      DWORD          dwReceiveDataSize;
8      DPNHANDLE      hBufferHandle;
9  } DPNMSG_RECEIVE, *PPDNMSG_RECEIVE;
```

- (1) 第 3 行：以字节来代表结构的大小。
- (2) 第 4 行：送出消息的玩家 ID。
- (3) 第 5 行：送出消息的玩家背景值。
- (4) 第 6 行：该消息的实际资料。
- (5) 第 7 行：返回 pReceiveData 的大小。
- (6) 第 8 行：pReceiveData 缓冲区的控制。

消息接收的部分，倒是不必特别利用 DirectPlay 的方法，只要使用 DirectPlay 响应函数所得到的消息并且加以分析，便可以轻易地得到所要的资料了。

## 5. 连接中断

中断正在联机中的网络连接是所有网络机制中最为简单的一种，因为中断联机不必事先取得对方的请求或响应，便可以直接切断网络之间的联机。所以在此，各位只要利用“Close”的方法就可以中断玩家与服务器端或点对点拓扑的联机了。

## 11.1.5 联机程序范例介绍

下面来编写一组具有服务器（Server）与客户端（Client）的文字简易对话应用程序。为了方便读者快速了解 DirectPlay 的工作原理，接下来的两个范例程序不采用窗口模式的架构，以省去一些建构上的麻烦。

首先来建构对话联机应用程序的服务器（Server）端，如范例 11\_1（见随书光盘）所示。

首先声明 3 个服务器程序必要的对象。如下列所示：

```
1  IDirectPlay8Server*      g_pDPSServer = NULL;
2  IDirectPlay8Address*     g_pDeviceAddress = NULL;
3  IDirectPlay8Address*     g_pHostAddress = NULL;
```

其中分别定义出服务器对象、设备地址、本机地址 3 个必要对象。

接下来编写初始化 DirectPlay 的对象程序。

### 程序代码

```
1  HRESULT InitDirectPlay()
```

```

2  {
3      HRESULT hr = S_OK;
4      // 建立 IDirectPlay8Server 对象
5      if( FAILED( hr = CoCreateInstance( CLSID_DirectPlay8Server, NULL,
6                                          CLSCTX_INPROC_SERVER,
7                                          IID_IDirectPlay8Server,
8                                          (LPVOID*) &g_pDPsServer ) ) )
9      {
10         printf("Failed Creating the IDirectPlay8Peer Object: 0x%X\n", hr);
11         goto LCleanup;
12     }
13     // 初始化 DirectPlay
14     if( FAILED( hr = g_pDPsServer->Initialize(NULL, DirectPlayMessageHandler, 0) ) )
15     {
16         printf("Failed Initializing DirectPlay: 0x%X\n", hr);
17         goto LCleanup;
18     }
19     // 建立以 TCP/IP 为通讯协议的 Server 端
20     if( FALSE == IsServiceProviderValid(&CLSID_DP8SP_TCP/IP) )
21     {
22         hr = E_FAIL;
23         printf("Failed validating CLSID_DP8SP_TCP/IP");
24         goto LCleanup;
25     }
26 LCleanup:
27     return hr;
28 }

```

**程序说明**

- (1) 第 5~8 行：建立 IDirectPlay8Server 对象。
  - (2) 第 14 行：初始化 DirectPlay 对象，并且将消息函数“DirectPlayMessageHandler”的指针输入其中。
  - (3) 第 20 行：建立以 TCP/IP 为通讯协议的 Server 端。
- 在上述程序代码中，将 DirectPlay 接收消息的响应函数定义为“DirectPlayMessageHandler”，接下来看程序代码的编写方法。

**程序代码**

```

1  HRESULT WINAPI DirectPlayMessageHandler(PVOID pvUserContext,
2                                          DWORD dwMessageId, PVOID pMsgBuffer)
3  {
4      HRESULT hr = S_OK;
5      switch (dwMessageId)
6      {
7          case DPN_MSGID_RECEIVE:
8              {
9                  PDPNMSG_RECEIVE pMsg;
10                 pMsg = (PDPNMSG_RECEIVE) pMsgBuffer;
11                 printf("\n 接收到的消息为: %S\n", (WCHAR*)pMsg->pReceiveData);

```

```

12         break;
13     }
14 }
15 return hr;
16 }

```

## 程序说明

- (1) 第 5 行: 根据 DirectPlay 触发的事件而运行特定的操作。
  - (2) 第 7 行: 当 directPlay 从网络上接收到消息时就会触发。
  - (3) 第 9~12 行: 将接收到的消息显示在屏幕中。
- 在 DirectPlay 初始化成功后, 接着便可以开始建立其设备的地址了。

## 程序代码

```

1 HRESULT CreateDeviceAddress()
2 {
3     HRESULT hr = S_OK;
4     // 建立 IDirectPlay8Address 设备的地址
5     if( FAILED( hr = CoCreateInstance(CLSID_DirectPlay8Address, NULL,
6                                     CLSCTX_INPROC_SERVER,
7                                     IID_IDirectPlay8Address,
8                                     (LPVOID*) &g_pDeviceAddress ) ) )
9     {
10         printf("IDirectPlay8Address 对象建立失败: 0x%X\n", hr);
11         goto LCleanup;
12     }
13     // 设定设备地址
14     if( FAILED( hr = g_pDeviceAddress->SetSP(&CLSID_DP8SP_TCPIP ) ) )
15     {
16         printf("设定设备地址失败: 0x%X\n", hr);
17         goto LCleanup;
18     }
19 LCleanup:
20     return hr;
21 }

```

## 程序说明

- (1) 第 5~8 行: 建立 IDirectPlay8Address 的对象。
  - (2) 第 14 行: 设定 IDirectPlay8Address 对象的设备地址。
- 将必要的对象都建构完成之后, 接下来只要让服务器具有等待服务的功能就可以了。

## 程序代码

```

1 HRESULT HostSession()
2 {
3     HRESULT hr = S_OK;
4     DPN_APPLICATION_DESC dpAppDesc;
5     DPN_PLAYER_INFO dpPlayerInfo;
6     WCHAR wszSession[128];
7     WCHAR wszName[] = L"Server";
8     ZeroMemory(&dpPlayerInfo, sizeof(DPN_PLAYER_INFO));

```

```

9    dpPlayerInfo.dwSize = sizeof(DPN_PLAYER_INFO);
10   dpPlayerInfo.dwInfoFlags = DPNINFO_NAME;
11   dpPlayerInfo.pwszName = wszName;
12   dpPlayerInfo.pvData = NULL;
13   dpPlayerInfo.dwDataSize = NULL;
14   dpPlayerInfo.dwPlayerFlags = 0;
15   if( FAILED( hr = g_pDPSTServer->SetServerInfo( &dpPlayerInfo, NULL, NULL,
16                                                    DPNSETSERVERINFO_SYNC ) ) )
17   {
18       printf("Failed Hosting: 0x%X\n", hr);
19       goto LCleanup;
20   }
21   printf("\n请输入本机名称.\n");
22   wscanf(L"%ls", wszSession);
23
24   ZeroMemory(&dpAppDesc, sizeof(DPN_APPLICATION_DESC));
25   dpAppDesc.dwSize = sizeof(DPN_APPLICATION_DESC);
26   dpAppDesc.dwFlags = DPNSESSION_CLIENT_SERVER;
27   dpAppDesc.guidApplication = g_guidApp;
28   dpAppDesc.pwszSessionName = wszSession;
29   if( FAILED( hr = g_pDPSTServer->Host(&dpAppDesc,
30                                         &g_pDeviceAddress, 1,
31                                         NULL, NULL,
32                                         NULL,
33                                         0 ) ) )
34   {
35       printf("本机名称失效: 0x%X\n", hr);
36       goto LCleanup;
37   }
38   else
39   {
40       printf("本机等待中...\n");
41   }
42
43 LCleanup:
44   return hr;
45 }

```

**程序说明**

- (1) 第 3~16 行: 建立服务器的信息。
- (2) 第 21~22 行: 让使用者输入服务器的本机名称。
- (3) 第 24~33 行: 设定服务器信息。

因为使用纯文字模式的架构, 所以只要将上述的主要程序代码放在 main() 函数中就可以实现所有的功能了, 如下所示。

**程序代码**

```

1  int main(int argc, char* argv[], char* envp[])
2  {

```



```

3     HRESULT          hr;
4     int              iUserChoice;
5     //初始化COM接口
6     CoInitializeEx(NULL, COINIT_MULTITHREADED);
7     // 初始化DirectPlay 接口
8     if( FAILED( hr = InitDirectPlay() ) )
9     {
10         printf("初始化DirectPlay 失败: 0x%X\n", hr);
11         goto LCleanup;
12     }
13     if( FAILED( hr = CreateDeviceAddress() ) )
14     {
15         printf("建立地址失败: 0x%X\n", hr);
16         goto LCleanup;
17     }
18     if( FAILED( hr = HostSession() ) )
19     {
20         printf("建立本机失败: 0x%X\n", hr);
21         goto LCleanup;
22     }
23     // 选择菜单
24     do
25     {
26         printf("请选择项目.\n1. 离开\n2. 传送消息\n");
27         scanf("%d", &iUserChoice);
28         if( iUserChoice == USER_SEND )
29         {
30             if( FAILED( hr = SendDirectPlayMessage() ) )
31             {
32                 printf("传送消息失败: 0x%X\n", hr);
33                 goto LCleanup;
34             }
35         }
36     } while (iUserChoice != USER_EXIT);
37
38 LCleanup:
39     CleanupDirectPlay();
40     // 释放COM接口
41     CoUninitialize();
42     return 0;
43 }

```

## 程序说明

- (1) 第7行: 初始化 DirectPlay 接口。
- (2) 第13行: 建立 DirectPlay 设备地址。
- (3) 第18行: 建立本机等待机制。
- (4) 第4~36行: 使用无穷循环来显示菜单, 如果使用者按下【1】, 则离开循环。

到这里已经成功地建构了一个简易的服务器应用程序。接下来, 则要让这个服务器应用程序不

但能够接收资料，而且能够传送资料，所以最后还要再编写一个函数，让这个函数可以为服务器程序传送资料。

#### 程序代码

```
1 HRESULT SendDirectPlayMessage()  
2 {  
3     HRESULT          hr = S_OK;  
4     DPN_BUFFER_DESC  dpnBuffer;  
5     WCHAR            wszData[256];  
6     printf("\n 输入字符串.\n");  
7     wscanf(L"%ls", wszData);  
8     dpnBuffer.pBufferData = (BYTE*) wszData;  
9     dpnBuffer.dwBufferSize = 2 * (wcslen(wszData) + 1);  
10    if( FAILED( hr = g_pDPSServer->SendTo(DPNID_ALL_PLAYERS_GROUP,  
11                                           &dpnBuffer,  
12                                           1,  
13                                           0,  
14                                           NULL,  
15                                           NULL,  
16                                           DPNSSEND_SYNC |  
17                                           DPNSSEND_NOLOOPBACK ) ) )  
18    {  
19        printf("送出资料失败: 0x%x\n", hr);  
20    }  
21    return hr;  
22 }
```

#### 程序说明

- (1) 第 6~7 行：让使用者可以输入欲传送的字符串。
- (2) 第 8~17 行：传送资料消息至网络。

#### 运行结果

程序运行结果如图 11-4~图 11-6 所示。

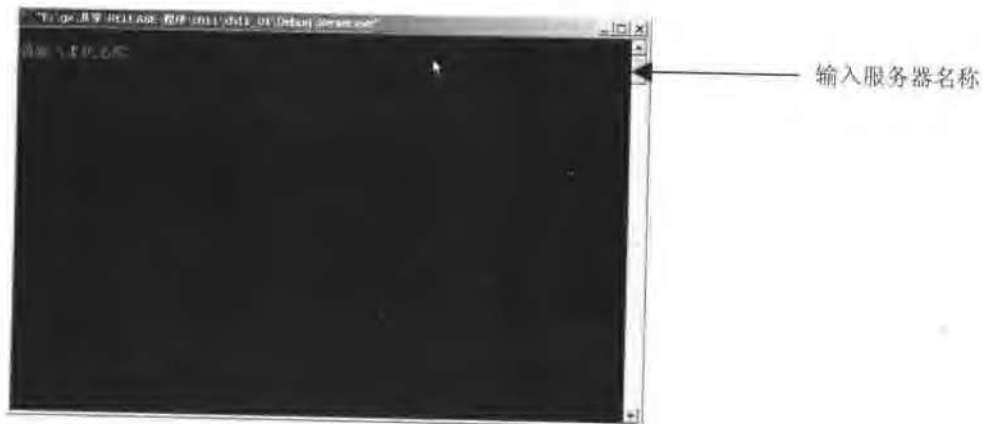


图 11-4

输入“1”，  
离开程序



图 11-5



服务器等待消息中...

图 11-6

在建立完成服务器的应用程序之后，接下来便可以开始编写客户端的应用程序了，如范例 ch11\_2（见随书光盘）所示。

其实客户端的程序代码与服务器的程序代码大同小异，惟一不同的是客户端应用程序必须先连上服务器的应用程序才可以运作。

在程序的开始部分，先来看一下 main 程序进入点的程序代码。

## 程序代码

```

1  int main(int argc, char* argv[], char* envp[])
2  {
3      HRESULT          hr;
4      int              iUserChoice;
5      CoInitializeEx(NULL, COINIT_MULTITHREADED);
6      if( FAILED( hr = InitDirectPlay() ) )
7      {
8          printf("DirectPlay 初始化失败: 0x%X\n", hr);
9          goto LCleanup;
10     }
11     InitializeCriticalSection(&g_csHostList);
12     if( FAILED( hr = CreateDeviceAddress() ) )

```

```

13  {
14      printf("建立设备地址失败: 0x%X\n", hr);
15      goto LCleanup;
16  }
17  if( FAILED( hr = EnumDirectPlayHosts() ) )
18  {
19      printf("列举本机失败: 0x%X\n", hr);
20      goto LCleanup;
21  }
22  if( FAILED( hr = ConnectToSession() ) )
23  {
24      printf("建立联机失败: 0x%X\n", hr);
25      goto LCleanup;
26  }
27  else
28  {
29      printf("\n 建立联机成功.\n");
30  }
31  do
32  {
33      printf("请选择菜单.\n1. Exit\n2. Send Data\n");
34      scanf("%d", &iUserChoice);
35      if( iUserChoice == USER_SEND )
36      {
37          if( FAILED( hr = SendDirectPlayMessage() ) )
38          {
39              printf("传送消息失败: 0x%X\n", hr);
40              goto LCleanup;
41          }
42      }
43  } while (iUserChoice != USER_EXIT);
44  LCleanup:
45      CleanupDirectPlay();
46      CoUninitialize();
47      return 0;
48  }

```

**程序说明**

(1) 第 3~16 行: 与服务器程序一样, 必须将 DirectPlay 必要的对象先初始化完成才能使用它。

(2) 第 17~30 行: 与服务器程序不同的地方就在这里。在这里, 必须先列举可用的服务器, 并且取得与它的联机。

(3) 第 31~42 行: 让使用者等待输入的菜单。

接下来看看与服务器不同的这两个函数里到底有何玄机。

**程序代码**

```

1  HRESULT EnumDirectPlayHosts()
2  {
3      HRESULT          hr = S_OK;

```

```

4   WCHAR          wszHost[128];
5   DPN_APPLICATION_DESC  dpAppDesc;
6   WCHAR*         pwszURL = NULL;
7   printf("\n 输入远程主机地址:\n");
8   wscanf(L"%ls", wszHost);
9   if( FAILED( hr = CreateHostAddress(wszHost) ) )
10  {
11      printf("建立远程主机失败: 0x%X\n", hr);
12      goto LCleanup;
13  }
14  ZeroMemory(&dpAppDesc, sizeof(DPN_APPLICATION_DESC));
15  dpAppDesc.dwSize = sizeof(DPN_APPLICATION_DESC);
16  dpAppDesc.guidApplication = g_guidApp;
17  if( FAILED( hr = g_pDPClient->EnumHosts(&dpAppDesc,
18                                          g_pHostAddress,
19                                          g_pDeviceAddress,
20                                          NULL, 0,
21                                          4,
22                                          0,
23                                          0,
24                                          NULL,
25                                          NULL,
26                                          DPNENUMHOSTS_SYNC ) ) )
27  {
28      printf("列举服务器失败: 0x%X\n", hr);
29      goto LCleanup;
30  }
31  LCleanup:
32  return hr;
33  }

```

### 程序说明

(1) 第 7~8 行: 让使用者输入欲联机的主机地址。

(2) 第 14~30 行: 列举远程服务器。

下面是服务器联机的函数程序代码。

### 程序代码

```

1   HRESULT ConnectToSession()
2   {
3       HRESULT          hr = E_FAIL;
4       DPN_APPLICATION_DESC  dpnAppDesc;
5       IDirectPlay8Address*  pHostAddress = NULL;
6       ZeroMemory(&dpnAppDesc, sizeof(DPN_APPLICATION_DESC));
7       dpnAppDesc.dwSize = sizeof(DPN_APPLICATION_DESC);
8       dpnAppDesc.guidApplication = g_guidApp;
9       EnterCriticalSection(&g_csHostList);
10  if( g_pHostList && SUCCEEDED(hr =
11      g_pHostList->pHostAddress->Duplicate(&pHostAddress) ) )
12  {

```

```
13     hr = g_pDPClient->Connect(&dpnAppDesc,  
14                               pHostAddress,  
15                               g_pDeviceAddress,  
16                               NULL,  
17                               NULL,  
18                               NULL, 0,  
19                               NULL,  
20                               NULL,  
21                               DPMCONNECT_SYNC);  
22     if( FAILED( hr))  
23         printf("联机失败: 0x%x\n", hr);  
24     }  
25     else  
26     {  
27         printf("本机地址失败: 0x%x\n", hr);  
28     }  
29     LeaveCriticalSection(&g_csHostList);  
30     SAFE_RELEASE(pHostAddress);  
31     return hr;  
32 }
```

#### 程序说明

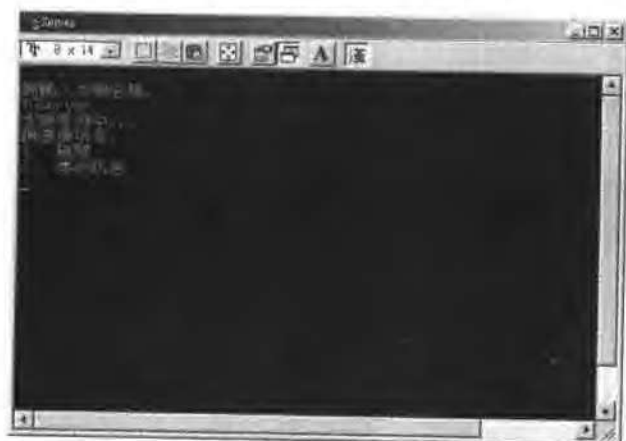
(1) 第 3~11 行: 列举并取得远程服务器的响应。

(2) 第 13~21 行: 取得远程服务器的联机。

其他的函数程序代码则与服务器没什么两样, 所以这里就不详加探讨了。  
在建构完服务器与客户端后, 接下来将这两个程序一起运行, 观看它们的效果。

#### 运行结果

程序运行结果如图 11-7~图 11-9 所示。



与上述的服务器操作一样, 先将服务器设定在等待中的状态。

图 11-7

输入远程服务器的地址，使得客户端可连上服务器。



图 11-8



在客户端上输入消息，并且传送至服务器。在服务器上便会收到客户端所传来的消息。

图 11-9

以上把基本的 DirectPlay 联机机制介绍完毕，相信各位对于 DirectPlay 应该有了相当的认识。最后，我们再来看看 DirectX 最后一项关于播放多媒体的开发函数库。

## 11.2 DirectShow 的多媒体功能

DirectX 开发函数库中的“DirectShow”功能，可以用来播放许多类型的媒体文件，如 AVI、MPG、MP3 等。下面先介绍有关 DirectShow 的基本流程架构。

### 11.2.1 DirectShow 的架构

第一步先来认识 DirectShow 中最重要的“过滤器”技术。“过滤器”的技术原理就是将资料流当做输入，然后在该资料流上进行单一的操作，最后将资料流输出。例如，DirectShow 的过滤器可以接收任何有效媒体文件（如 MPEG）资料流的输入，然后将它解压，最后再将压缩后的影像画面传送到显示卡上，如图 11-10 所示。



图 11-10

DirectShow 本身具有相当多的过滤器，它们可以组合在一起操作，也就是说，有可能一个过滤器的输出结果会变成另一个过滤器的输入。演示流程如图 11-11 所示。



图 11-11

DirectShow 还具有另外一种独特的组件，即过滤器管理员。顾名思义，它的工作就是管理过滤器，可以自动建立一个播放特定文件的过滤器顺序，如图 11-12 所示。

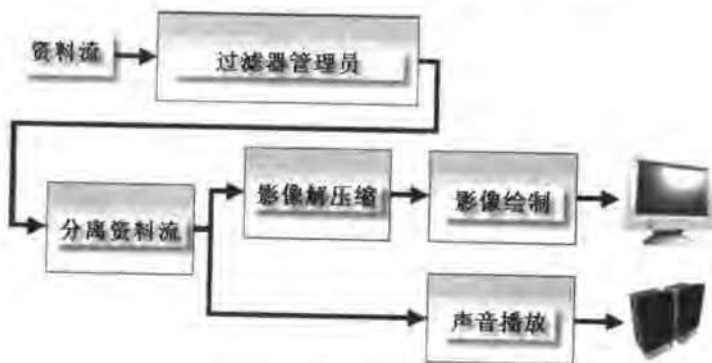


图 11-12

过滤器管理员的好处是可以自动建立一个过滤图，免去自己建构过滤图的麻烦。

将过滤器管理员建构好后，接下来便可以利用它所提供的方法来激活或停止多媒体文件的播放。

DirectShow 是一个架构非常复杂的组件，不过幸运的是它已经将这些复杂的建立过程与播放做得非常简单。所以目前很少有人会再去研究在多媒体文件的连接下，这些过滤器要如何结合媒体文件的技术了。一般来说，只要利用 `RenderFile` 与 `Run` 的方法就可以开始播放动画了。



## 11.2.2 播放影片功能

在了解 DirectShow 的架构之后,接着便可以开始动手编写播放动画的程序代码了,如范例 ch11\_3 (见随书光盘) 所示。

在此不需要额外的窗口设定,只需要利用 main() 作为程序的进入点,再利用 DirectShow 的技术便可以轻易地播放媒体动画文件了,如下所示。

### 程序代码

```

1  #include "stdafx.h"
2  #include <dshow.h>
3  int main()
4  {
5      IGraphBuilder *pGraph;
6      IMediaControl *pMediaControl;
7      IMediaEvent *pEvent;
8      CoInitialize(NULL);
9      // 建立 Media 对象
10     CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
11                     IID_IGraphBuilder, (void **)&pGraph);
12     pGraph->QueryInterface(IID_IMediaControl, (void **)&pMediaControl);
13     pGraph->QueryInterface(IID_IMediaEvent, (void **)&pEvent);
14     // 读取 "test.avi" 文件
15     pGraph->RenderFile(L"test.avi", NULL);
16     // 绘制动画
17     pMediaControl->Run();
18     // 等待动画播放结束
19     long evCode;
20     pEvent->WaitForCompletion(INFINITE, &evCode);
21     // 清除对象
22     pEvent->Release();
23     pMediaControl->Release();
24     pGraph->Release();
25     CoUninitialize();
26     return 0;
27 }
```

### 程序说明

- (1) 第 5~7 行: 声明 DirectShow 的必要对象, 如 IGraphBuilder 为绘图对象、IMediaControl 为媒体控制对象、IMediaEvent 为媒体事件对象。
- (2) 第 10~13 行: 建立基本绘图对象、媒体播放对象。
- (3) 第 15 行: 读取媒体文件。
- (4) 第 17 行: 播放媒体文件。
- (5) 第 19~20 行: 等待媒体文件的播放结束。
- (6) 第 22~25 行: 释放所有建立的媒体对象。

**运行程序**

程序运行结果如图 11-13 所示。



图 11-13

### 11.2.3 播放 MP3

事实上，也可以利用 DirectShow 播放 MP3 的文件。其实播放 MP3 并不需要再编写另外的程序代码来实现，只要利用范例 ch11\_2 中的代码即可。只要将范例 11\_3 中的第 15 行改成欲播放的 MP3 音乐文件，就可以轻轻松松听到 MP3 内所包含的音乐信息了。

前面已经介绍了 DirectPlay 与 DirectShow 的基本常识，相信各位可以利用之前讲述过的内容来编写网络联机与播放媒体文件的程序代码了。

## 课后重点整理

- DirectPlay 包含了网络游戏的基本架构，使得软件工程师可以快速上手并轻松使用它。
- DirectPlay 在 DirectX 系列中的主要功能是提供控制网络上数据传输的函数特性，以方便开发者的使用。
- 开发网络型的游戏时，可以利用 DirectPlay 所提供的网络函数库来进行开发或直接使用 Win32 本身所提供的 TCP/IP 的 API 函数。
- 一般来说，大多数的游戏会在游戏的第一个画面中，让玩家去选择一个适合自己的通讯服务协议（如 IPX、TCP/IP、串联、调制解调器等等），当要列举这些通讯服务协议时，可以调用 EnumServiceProviders 来完成。
- 在 DirectPlay 中可以借助 SetServerInfo 的方法（已有 DirectPlayServer 接口）或 SetPeerInfo 的方法（已有 DirectPlayPeer 接口）来设定玩家的名称或其他信息。
- 当玩家连接到游戏之后，可以利用 SendTo 的方法（服务器端或在点对点的拓扑中），或以 Send 的方法（客户端）将资料传送给其他计算机。
- 只要利用“Close”方法就可以中断玩家与服务器或点对点拓扑的联机了。
- “DirectShow”功能可以用来播放许多类型的媒体文件，如 AVI、MPG、MP3 等。
- DirectShow 中“过滤器”的技术原理就是将资料流当做输入，然后在该资料流上进行单一的操作，最后将资料流输出。例如 DirectShow 的过滤器可以接收任何有效媒体文件（如

MPEG) 资料流的输入, 然后将它解压, 最后再将未经压缩的影像画面传送到显示卡上。

- DirectShow 中过滤器管理员的工作是管理过滤器, 可以通过过滤器管理员自动建立一个播放特定文件的过滤器顺序。
- 过滤器管理员的好处是帮助自动建立一个过滤图, 免去自己建构过滤图的麻烦。

## 课后习题

- (1) DirectPlay 可以支持哪两种网络拓扑的方式?
- (2) 请简述 DirectShow 的主要功能。
- (3) 利用 DirectPlay 的函数库来开发游戏之前, 必须先规划出一系列的使用者基本程序, 这个基本程序应该如何分类? 试简单说明之。
- (4) 试简述 DirectPlay 的使用时机。

## 第 12 章 小游戏设计实例

### 12.1 俄罗斯方块游戏轻松做

在游戏设计史上,相信大家对于俄罗斯方块游戏一定都记忆犹新,这个玩法简单的迷你游戏席卷了整个世界。各种游戏平台、大型电脑、家用游戏机等,甚至连掌上型的小游戏机都有它的身影。几乎可以说只要有计算机的地方,就有俄罗斯方块。它不只玩法简单,设计理念也很简单。

范例 ch12\_Tetris (见附书光盘) 仅使用 Direct Graphics 及几张简单的图完成,本节就来看看这款游戏的设计方法。

俄罗斯方块的相关游戏规则无需详加介绍了。重新编译范例后,可以看到如图 12-1 所示的运行画面。

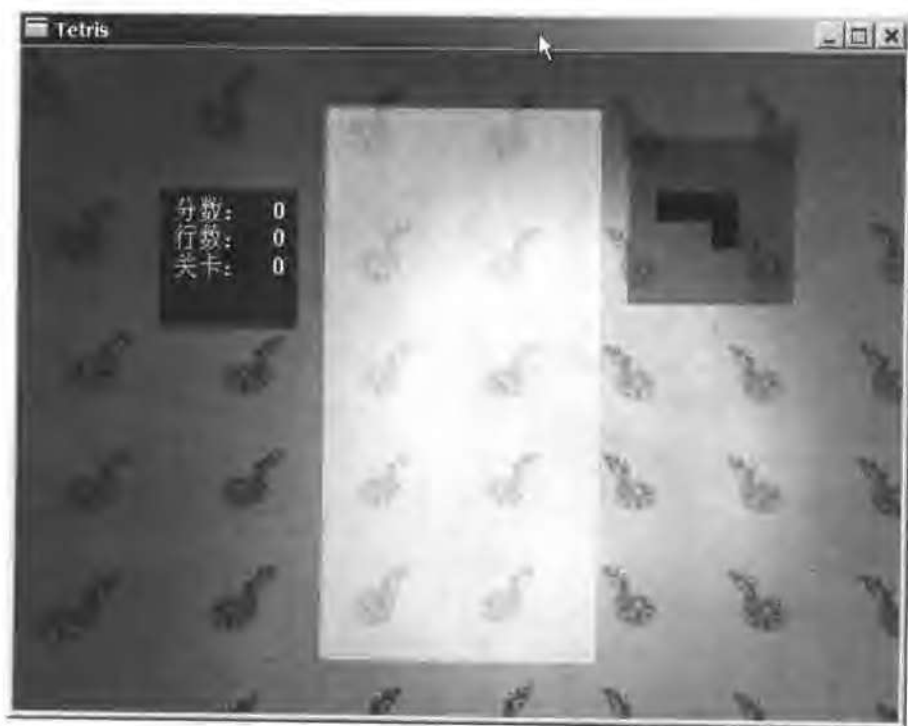


图 12-1

按【Z】键,即可开始玩游戏。游戏界面如图 12-2 所示。

用方向键来左右移动方块,【Z】键旋转方块,右方为下一个出现的方块。

ch12\_Tetris 的设计系统可分为三个部分,依次为:消息循环、玩家输入及游戏循环。

一般的 MFC 架构中,通常借助 Windows 传递来的消息分析处理后更新画面,但这一方式对游戏而言不够直接,需要利用更直接的方式。

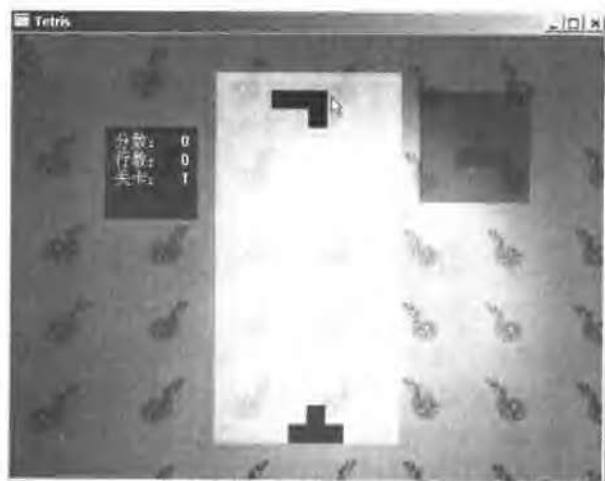


图 12-2

函数 `canvasApp::Run()` 即是取代传统 Windows 消息循环的方式。

程序代码: `canvasApp.cpp`

```

1  int canvasApp::Run()
2  {
3      MSG msg ;
4      while(true)
5      {
6          if( PeekMessage( &msg , 0 , 0 , 0 , PM_REMOVE ))
7          {
8              TranslateMessage( &msg );
9              DispatchMessage( &msg );
10             if( msg.message == WM_QUIT )
11                 return msg.wParam ;
12             else
13             {
14                 //操作更新
15                 diRun();
16                 //响应运行
17                 if( app_Proc )
18                     app_Proc->Run();
19             }
20         }
21         return 0 ;
22     }

```

程序说明:

(1) 第 6 行: 用 `PeekMessage` 取代 `GetMessage`, 两者差别在于 `GetMessage` 为“等待到取得消息为止”, 而 `PeekMessage` 为“不等待”, 这一函数只要 Windows 传送消息就优先处理它, 如果没有则运行游戏循环。

(2) 第 15 行: 接收目前的键盘状态。

(3) 第 17 行: 游戏循环的处理, 调用游戏循环骨架的函数 `Tetris:Updata()` 来运行。

## 1. 键盘与玩家操作方法

利用 DirectInput 取得玩家输入。游戏循环一开始,先调用 diRun(),直接取得目前键盘状态并转换成玩家输入的按键。这样可以让玩家不论是用键盘、鼠标还是控制器输入,都可转换成共同的格式,只要在游戏中判断格式的内容就可以了。操作流程如图 12-3 所示。



图 12-3

可以在 di.h 看到为玩家定义的操作状态。

```

1  typedef struct _DI_DEVICE_DATA
2  {
3      DWORD      _time ;      //经过时间
4      int        News4 ;      //点一下方向
5      int        News4ing ;    //按住方向
6      BYTE       Bn[4] ;      //按一下按键
7      BYTE       Bning[4] ;    //按住按键
8  }DI_DEVICE_DATA ;
  
```

设定标准的键盘输入情形,支持 1 个方向键及 4 个按键,并且把循环更新时间定义在结构中。

“键盘”转换为“玩家操作”的代码在函数 diRun 中,转换过程可分为“循环时间”→“方向转换”→“按键转换”三个步骤。

### 程序代码: di.cpp

```

1  void diRun()
2  {
3      int i ;
4      int news ;
5      DWORD tNow ;
6      const BYTE key[4] = { DIK_Z , DIK_X , DIK_C , DIK_V , } ;
7      //时间
8      do
9      {
10         tNow = timeGetTime();
11         di_DDD._time = tNow - di_TimeBuffer ;
12     }while( di_DDD._time < 15 );
13     if( di_DDD._time > 40 )di_DDD._time = 40 ;
14     di_TimeBuffer = tNow ;
15     //取得键盘状态
16     if( di_Key )
17         if( di_Key->GetDeviceState( 256 ,(LPVOID)&di_KeyData )== DI_OK )
18         {
19             //方向键
20             if( di_KeyData[ DIK_DOWN] & 0x80 )
21                 news = 1 ;      //下
  
```

```

22     else if( di_KeyData[ DIK_UP ] & 0x80 )
23         news = 5 ;           //上
24     else if( di_KeyData[ DIK_LEFT ] & 0x80 )
25         news = 7 ;           //左
26     else if( di_KeyData[ DIK_RIGHT ] & 0x80 )
27         news = 3 ;           //右
28     else news = 0 ;
29 //方向移动中
30     if( news == di_DDD.News4ing )
31     {
32         if( di_DDD.News4ing )
33             di_DDD.News4 = 0 ;
34         else
35         {
36             di_DDD.News4 = news ;
37             di_DDD.News4ing = news ;
38         }
39     }else
40     {
41         di_DDD.News4 = news ;
42         di_DDD.News4ing = news ;
43     }
44 //按键
45     for( i = 0 ; i < 4 ; i++ )
46     {
47         if( di_KeyData[ key[i] ] & 0x80 )
48         {
49             if( di_DDD.Bning[i] )
50                 di_DDD.Bn[i] = false ;
51             else
52             {
53                 di_DDD.Bn[i] = true ;
54                 di_DDD.Bning[i] = true ;
55             }
56         }else
57         {
58             di_DDD.Bn[i] = false ;
59             di_DDD.Bning[i] = false ;
60         }
61     }
62 }
63 }

```

## **程序说明:**

(1) 第 8~14 行: 计算游戏循环时间, 用 `timeGetTime` 取得目前系统时间, 减去上次取得的时间, 即为经过的时间, 如果其介于 0.015~0.04 秒之间 (约 30~60fps)。

(2) 第 20~43 行: 键盘方向转换为玩家操作方向, 下右上左依序为 1、3、5、7, 0 为玩家没有按方向键, 如有兴趣也可以改为 8 个方向。

(3) 第45~61行: 把键盘按键转换为玩家操作按键, 程序第6行时定义按键为【Z】、【X】、【C】、【V】。

## 2. 游戏循环结构

这里将使用 C++ 的继承概念, 如果不熟悉 C++ 的继承概念也没关系, 继续往下阅读。可以在 appProc 中看到对游戏循环的对象的声明。

```
1 class appProc
2 {
3 public:
4     virtual void Render();
5     virtual void Run();
6 };
```

定义 Run 及 Render 两个虚拟 (virtual) 函数来处理游戏的流程与绘制, 游戏中所有处理程序继承对象 appProc, 分别为准备开始 (TetrisIs)、游戏进行 (Tetris)、游戏结束 (TetrisGameOver) 三种。只要调用基类函数 appProc::Run() 即可处理游戏的进行。流程如图 12-4 所示。

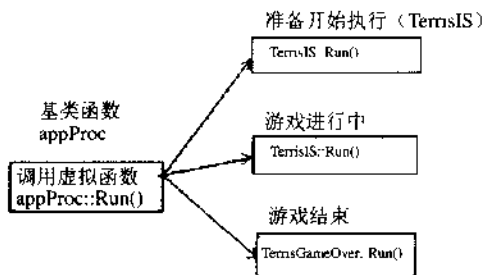


图 12-4

对这三个对象分别编写三个函数进行转换, 即 appTetrisGameOverRun (游戏结束)、appTetrisIsRun (准备开始) 和 appTetrisRun (游戏进行中)。在窗口建立消息 (WM\_CREATE) 时, 可以调用函数 appTetrisIsRun() 将游戏基类导向准备开始对象, 这三个函数的内容大同小异, 就是把处理程序对象设定给游戏框架, 让框架中的虚拟函数进行操作。

```
1 TetrisGameOver app_TSGameOver ;
2 void appTetrisGameOverRun()
3 {
4     app_Proc = &app_TSGameOver ;
5 }
```

这三个对象只有在游戏处理 (Run) 与绘制 (Render) 中有些差异。下面主要以游戏进行中的对象 Tetris 为主, 对象准备开始 (TetrisIs) 及游戏结束 (TetrisGameOver) 都直接调用 Tetris::Render() 进行背景绘制, 并输出一些文字, 也可以贴图或接口用来替换。

## 3. 俄罗斯方块的进行方式

俄罗斯方块的游戏大致可分为以下 9 个要素。

(1) 每四个小方块可组合成一组方块群。



- (2) 从七组方块群中, 随机取一组, 并且告诉玩家下一组方块是什么。
- (3) 在  $10 \times 20$  的游戏框中, 方块由上方慢慢落下。
- (4) 玩家可按【←】、【→】键移动方块, 按【↓】键加速方块落下的速度, 或按【Z】键旋转方块, 所有的方块要在游戏框中。
- (5) 方块落下时, 四个小方块中的一个到达游戏框最底或是下一格有方块时, 即停止移动, 并将方块移到游戏框底。
- (6) 如果同一行的 10 格中全部填满小方块, 即可消除该行, 上面的方块以行为单位, 由上一行一行一行地往下移。
- (7) 当消除的行数越多时, 落下的速度就越快。
- (8) 如果方块停住时有方块未在游戏框内, 游戏结束。
- (9) 生成一个新的方块, 再回到步骤 2。

根据以上的要求, 定义下面的结构来存放游戏资料, 可以在 appProc 中看到以下结构。

```

1  typedef struct _TETRIS_DATA
2  {
3      BYTE Data[10][20];           //数据资料
4      POINT NowBox[4];             //目前方块
5      POINT NowPoint;
6      int BoxNextIndex;            //下一个方块
7      int BoxNowIndex;             //目前方块
8      int TimeNextGrid;            //方块落下时间
9      int TimeGridNow;             //方块落下目前时间
10     int sLV;                      //关卡
11     int sNLV;                     //下一关要消的行数
12     int sScore;                   //得分
13     int sDel;                     //已消除行数
14 }TETRIS_DATA;
15 TETRIS_DATA ts_DA;
16 //背景
17 d3dTexture ts_Back;
18 //方块
19 d3dTexture ts_Box[7];
    
```

除游戏的资料外还定义了对象 d3dTexture 存放图片 (一张背景图及 7 个小方块图)。

#### 4. 方块群的组成

这个游戏中最主要的组件是由四个小方块组合成的方块群, 排列组合后可以组成以下 7 组完全不同的方块群, 如图 12-5 所示。

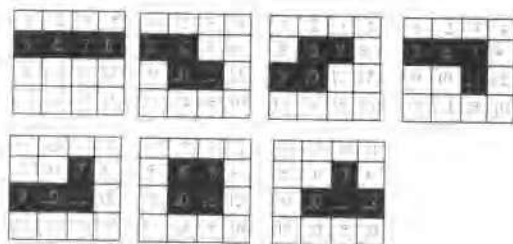


图 12-5

程序中各方块群的位置如下所示。

```

1  static const POINT app_TetrisBox[7][4] =
2  {
3      /*0
4          □□□□
5      */({ 0, 1 }, { 1, 1 }, { 2, 1 }, { 3, 1 }},
6      /*1
7          □□
8          □□
9      */({ 0, 1 }, { 1, 1 }, { 2, 1 }, { 2, 2 }},
10     /*2
11         □□
12         □□
13     */({ 0, 2 }, { 1, 2 }, { 1, 1 }, { 2, 1 }},
14     /*3
15         □□□
16         □
17     */({ 0, 1 }, { 1, 1 }, { 2, 1 }, { 2, 2 }},
18     /*4
19         □
20         □□□
21     */({ 0, 2 }, { 1, 2 }, { 2, 2 }, { 2, 1 }},
22     /*5
23         □□
24         □□
25     */({ 1, 1 }, { 2, 1 }, { 2, 2 }, { 1, 2 }},
26     /*6
27         □
28         □□□
29     */({ 2, 1 }, { 1, 2 }, { 2, 2 }, { 3, 2 }},
30 };

```

## 5. 游戏初始化

所有的组件都准备好后, 开始进行游戏环境的初始化工作, 让玩家随时可以开始游戏。初始化过程如下所示。

**程序代码: appProc.cpp**

```

1  void Tetris::Updata()
2  {
3      int i ;
4      char ch[256] ;
5      //开始
6      memset( &ts_DA , 0 , sizeof( ts_DA ));
7      //加载背景
8      ts_Back.Create( "背景.tga" );
9      //载入方块
10     for( i = 0 ; i < 7 ; i++ )

```

```

11 {
12     wsprintf( ch , "方块%2.2d.bmp" , i );
13     ts_Box[i].Create( ch );
14 }
15 //第一个方块
16 ts_DA.BoxNextIndex = rand()%7 ;
17 GetBox();
18 //时间
19 ts_DA.TimeNextGrid = 560 ;
20 //绘制模式
21 d3d_Device->SetTextureStageState( 0 , D3DTSS_ALPHARG1 , D3DTA_TEXTURE );
22 d3d_Device->SetTextureStageState( 0 , D3DTSS_ALPHARG2 , D3DTA_DIFFUSE );
23 d3d_Device->SetTextureStageState( 0 , D3DTSS_COLORARG1 , D3DTA_TEXTURE );
24 d3d_Device->SetTextureStageState( 0 , D3DTSS_COLORARG2 , D3DTA_DIFFUSE );
25 }

```

## 程序说明

- (1) 第 6 行：将游戏资料清空。
- (2) 第 8~14 行：加载背景图及小方块图。
- (3) 第 16~17 行：取得第一个方块。
- (4) 第 21~24 行：Direct Graphics 的材质操作基台的设定。

Tetris::GetBox()为取得方块的函数，说明如下所示。

## 程序代码：appProc.cpp

```

1 void Tetris::GetBox()
2 {
3     int i ;
4     //定控制基准点初始位置
5     ts_DA.NowPoint.x = 3 ;
6     ts_DA.NowPoint.y = -3 ;
7     //定四个方块位置
8     ts_DA.BoxNowIndex = ts_DA.BoxNextIndex ;
9     for( i = 0 ; i < 4 ; i++ )
10 {
11     ts_DA.NowBox[i].x = app_TetrisBox[ ts_DA.BoxNowIndex ][i].x ;
12     ts_DA.NowBox[i].y = app_TetrisBox[ ts_DA.BoxNowIndex ][i].y ;
13 }
14 //下一个方块
15 ts_DA.BoxNextIndex = rand()%7;
16 //时间初始化
17 ts_DA.TimeGridNow = ts_DA.TimeNextGrid ;
18 }

```

## 程序说明

- (1) 第 5~6 行：方块基准点为整个方块群的原点，所有小方块均由方块基准点算出实际位置，(3,-3) 为游戏框上方中间。
- (2) 第 8~13 行：取得方块。
- (3) 第 15 行：随机数取得下一个方块。

(4) 第17行: 定义方块落下的时间。

## 6. 游戏绘制

当游戏初始化完成后, 先来看游戏的画面。

```
1 void Tetris::Render()
2 {
3     //画对象
4     RenderBack();
5     //成像
6     d3d_Device->Present( NULL , NULL , NULL , NULL );
7 }
```

将游戏分循环分为3个对象, 如果每个对象都要重写绘制底图的函数就会比较麻烦。所以绘制时可以把“绘制”与“翻页”的工作分开, 以便其他对象能够随时调用并不用重新填写, 并且可以随时需要进行更新。

游戏的绘制函数如下:

```
1 void Tetris::RenderBack()
2 {
3     d3dTexture d3dRt ;
4     //清空
5     d3dClear();
6     //开始绘制
7     d3d_Device->BeginScene();
8     d3d_Device->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_SRCALPHA );
9     d3d_Device->SetRenderState( D3DRS_DESTBLEND , D3DBLEND_INVSRCALPHA );
10    d3d_Device->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
11    d3d_Device->SetTextureStageState( 0 , D3DTSS_ALPHAOP , D3DTOP_SELECTARG2 );
12    //画背景
13    ts_Back.BltFast( 0 , 0 );
14    //画底框
15    d3dRt.BltFast( 220 , 40 , 420 , 440 , 0x8FFFFFFF );
16    //文字框
17    d3dRt.BltFast( 100 , 100 , 200 , 200 , 0x8F000000 );
18    //下个方块框
19    d3dRt.BltFast( 440 , 60 , 560 , 180 , 0x4F000000 );
20    //画方块
21    DrawNowBox();
22    d3d_Device->EndScene();
23    //绘制文字
24    DrawText();
25 }
```

### 程序说明

- (1) 第7~11行: 绘制基台操作设定。
- (2) 第13~22行: 游戏画面的绘制。
- (3) 第24行: 游戏中的说明文件的绘制。

整个绘制工作分为三部分，分别为游戏框、方块及游戏信息绘制，如图 12-6 所示。



图 12-6

游戏中经常需要画单元格方块，因此将绘制工作独立出来，只要输入格位及方块代码即可，方块绘制函数如下所示。

```

1 void Tetris::DrawBox(int gx, int gy, int index)
2 {
3     int x, y;
4     if(( index >= 0 ) && ( index < 7 ))
5     {
6         x = gx * 20 + 220;
7         y = gy * 20 + 40;
8         ts_Box[index].BitFast( x, y );
9     }
10 }
    
```

#### 程序说明

- (1) 第 4 行：判断该方块代码是否在设定的材质里。
- (2) 第 6~7 行：设定单元格为 20×20，游戏框左上角为 (220,40)，计算屏幕位置。
- (3) 第 8 行：将图贴到屏幕上。

下面有三种方块要进行绘制，统一在函数 Tetris::DrawNowBox() 中进行。

```

1 void Tetris::DrawNowBox()
2 {
3     int i, j;
4     int gx, gy;
5     BYTE index;
6     //画底框方块
7     for( i = 0; i < 10; i++ )
8     for( j = 0; j < 20; j++ )
9         if( index = ts_DA.Data[i][j] )
10            DrawBox( i, j, index - 1 );
    
```

```

11 //画目前方块
12 for( i = 0 ; i < 4 ; i++ )
13 {
14     gx = ts_DA.NowBox[i].x + ts_DA.NowPoint.x ;
15     gy = ts_DA.NowBox[i].y + ts_DA.NowPoint.y ;
16     if( IsInRect( gx , gy ))
17         DrawBox( gx , gy , ts_DA.BoxNowIndex );
18 }
19 //画下一个方块
20 for( i = 0 ; i < 4 ; i++ )
21     DrawBox( app_TetrisBox[ts_DA.BoxNextIndex][i].x + 12 , app_TetrisBox
                [ts_DA.BoxNextIndex][i].y + 2 , ts_DA.BoxNextIndex );
22 }

```

#### 程序说明

- (1) 第7~10行：用循环的方式绘制游戏框内的方块，如果该格为0代表没有方块。
  - (2) 第12~18行：根据方块基准点取得实际方块的位置，并判定是否在游戏框内，从而进行绘制。
  - (3) 第20~21行，在游戏框右方画下一组方块。
- 绘制的最后一件工作就是将目前游戏的信息告诉玩家，可使用对象 d3dHdc 来取得 Direct Graphics 的绘图装置，再输出一些文字。

```

1 void Tetris::DrawText()
2 {
3     int len ;
4     char ch[256] ;
5     //文字
6     d3dHdc hdc ;
7     SetBkMode( hdc , 1 );
8     SetTextColor( hdc , RGB( 255 , 255 , 255 ));
9     //目前分数
10    len = wsprintf( ch , "分数: %d" , ts_DA.sScore );
11    TextOut( hdc , 110 , 105 , ch , len );
12    //已消行数
13    len = wsprintf( ch , "行数: %d" , ts_DA.sDel );
14    TextOut( hdc , 110 , 125 , ch , len );
15    //关卡
16    len = wsprintf( ch , "关卡: %d" , ts_DA.sLV );
17    TextOut( hdc , 110 , 145 , ch , len );
18 }

```

#### 程序说明

- (1) 第6行：取得 Direct Graphics 的绘图装置 (HDC)。
- (2) 第7~8行：设定文字背景模式及颜色。
- (3) 第10~17行：绘制游戏信息。

#### ● 方块落下

方块落下，即随着时间推移，方块一格一格的下落。在游戏中，方块落下程序如下：

```

1 void Tetris::BoxDown()
2 {
3     int i ;
4     int gx , gy ;
5     //落下时间
6     if( di_DDD.News4ing == 1 )
7         ts_DA.TimeGridNow -= ( di_DDD._time * 5 );
8     else
9         ts_DA.TimeGridNow -= ( di_DDD._time );
10    //成功落下
11    while( ts_DA.TimeGridNow < 0 )
12    {
13        ts_DA.TimeGridNow += ts_DA.TimeNextGrid ;
14        if( IsDown( ts_DA.NowPoint.x , ts_DA.NowPoint.y + 1 ))
15            ts_DA.NowPoint.y ++ ;
16        else
17        {
18            //停下来的方块若未进入格子内, 代表游戏结束
19            for( i = 0 ; i < 4 ; i++ )
20                if( !IsInRect( ts_DA.NowBox[i].x + ts_DA.NowPoint.x ,
21                    ts_DA.NowBox[i].y+ ts_DA.NowPoint.y ))
22                {
23                    appTetrisGameOverRun();
24                    return ;
25                }
26            //资料到后缓冲区
27            for( i = 0 ; i < 4 ; i++ )
28            {
29                gx = ts_DA.NowBox[i].x + ts_DA.NowPoint.x ;
30                gy = ts_DA.NowBox[i].y + ts_DA.NowPoint.y ;
31                ts_DA.Data[gx][gy] = ts_DA.BoxNowIndex + 1 ;
32            }
33            //判断是否可消除
34            ListDelIs();
35            //取下一个方块
36            GetBox();
37            //结束
38            return ;
39        }
40    }

```

## 程序说明

- (1) 第 6~9 行: 将落下的时间减去经过的时间。如果玩家按【↓】键, 将经过的时间乘 5 以更快的方式向下掉。
- (2) 第 11 行: 判定方块是否已可以下落。
- (3) 第 14~15 行: 判定四个小方块的下一格是否有空间, 如果有, 将方块基准点往下移。
- (4) 第 19~24 行: 判定这四个小方块是否超出游戏框内, 如果有, 则代表游戏失败, 将游戏

循环转到游戏结束对象 (TetrisGameOver)。

(5) 第 26~31 行: 将目前的方块位置写入游戏框中。

(6) 第 33 行: 调用 Tetris::ListDells()进行方块删除。

(7) 第 35 行: 生成一个新的方块, 继续进行游戏

方块删除函数说明如下:

```

1  void Tetris::ListDelIs()
2  {
3      //分数
4      static const int Score[] = { 0,100, 300, 1000, 4000 };
5      int i, j, k, num;
6      int list[20];
7      //判断横向是否全部有方块
8      num = 0;
9      memset( list, 0, sizeof( list ));
10     for( j = 19; j >= 0; j-- )
11         for( i = 0; i < 10; i++ )
12             if( ts_DA.Data[i][j] )
13                 list[j] ++;
14     //判断是否填满行
15     for( j = 0; j < 20; j++ )
16         if( list[j] == 10 )
17         {
18             //消去该行
19             for( i = 0; i < 10; i++ )
20                 ts_DA.Data[i][j] = 0;
21             //消去行加1
22             num ++;
23         }
24     if( num == 0 )
25         return;
26     //方块落下
27     for( i = j = 19; ( j >= 0 ); j-- )
28     {
29         //落下
30         for( ; i >= 0; i-- )
31             if( list[i] != 10 )
32             {
33                 //如果两者不同行,就由上向下移动
34                 if( i != j )
35                     for( k = 0; k < 10; k++ )
36                     {
37                         ts_DA.Data[k][j] = ts_DA.Data[k][i];
38                         ts_DA.Data[k][i] = 0;
39                     }
40                 //行数上升
41                 i--;
42                 break;

```



```

43         }
44     }
45     //记录分数
46     ts_DA.sScore += Score[num] ;
47     ts_DA.sDel += num ;
48     ts_DA.sNLV -= num ;
49     if( ts_DA.sNLV <= 0 )
50     {
51         ts_DA.sNLV = 6 ;
52         ts_DA.sLV ++ ;
53         if( ts_DA.sLV < 10 )
54             ts_DA.TimeNextGrid = 60 + ( 10 - ts_DA.sLV ) * 50 ;
55     }
56 }

```

## 程序说明

- (1) 第 4 行：定义消去一行给玩家一点得分。
- (2) 第 8~13 行：计算每一行中所有的方块数。
- (3) 第 15~25 行：判定同一行中的方块是否满 10 个，如果满了，则表示该行可以被删除，将该行清空。
- (4) 第 27~44 行：以行为单位由上向下移动。
- (5) 第 46~55 行：计算得分并判断是否晋级。

## 7. 方块的移动

前面曾经提到，方块移动并非实际上移动每一个方块，而是更改方块的基准点。所以在移动时，只要判定移动的方向是否已超出游戏框，或是否有其他的方块，最后将基准点加减 1 即可。

```

1  void Tetris::Move()
2  {
3      if( di_DDD.News4 == 3 )          //右移
4      {
5          if( IsDown( ts_DA.NowPoint.x + 1 , ts_DA.NowPoint.y ))
6              ts_DA.NowPoint.x ++ ;
7      }else if( di_DDD.News4 == 7 )    //左移
8      {
9          if( IsDown( ts_DA.NowPoint.x - 1 , ts_DA.NowPoint.y ))
10             ts_DA.NowPoint.x -- ;
11     }
12 }

```

## 程序说明

- (1) 第 3~6 行：判定玩家是否按了【→】键，并判断是否可右移，如果是，将方块基准点加 1。
- (2) 第 7~10 行：判定玩家是否按了【←】键，并判断是否可右移，如果是，将方块基准点减 1。

## 8. 方块的旋转

旋转简单的说就是以方块群的中心为基准旋转, 计算新的位置, 这里使用索引法计算新的位置。转旋后的各方块位置如图 12-7 所示。

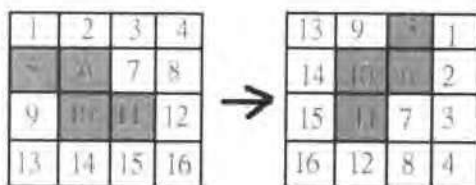


图 12-7

旋转的程序代码如下:

```

1  void Tetris::Rotation()
2  {
3      //转换矩阵
4      static const POINT ro[4][4] =
5      {
6          {{ 3, 0 }, { 2, 0 }, { 1, 0 }, { 0, 0 }},
7          {{ 3, 1 }, { 2, 1 }, { 1, 1 }, { 0, 1 }},
8          {{ 3, 2 }, { 2, 2 }, { 1, 2 }, { 0, 2 }},
9          {{ 3, 3 }, { 2, 3 }, { 1, 3 }, { 0, 3 }},
10     };
11     int i ;
12     int gx , gy ;
13     POINT po[4] ;
14     //如果未按下第一键, 表示不旋转
15     if( !di_DDD.Btn[0] )
16         return ;
17     //计算转换后位置
18     for( i = 0 ; i < 4 ; i++ )
19     {
20         po[i].x = ro[ ts_DA.NowBox[i].x ][ ts_DA.NowBox[i].y ].x ;
21         po[i].y = ro[ ts_DA.NowBox[i].x ][ ts_DA.NowBox[i].y ].y ;
22         //判定是否在矩阵内, 如果不在矩阵内不转
23         gx = po[i].x + ts_DA.NowPoint.x ;
24         gy = po[i].y + ts_DA.NowPoint.y ;
25         if( !IsInRect( gx , gy ))
26             return ;
27         //判定是否有方块, 如果有则方块不转
28         else if( ts_DA.Data[gx][gy] )
29             return ;
30     }
31     //旋转成功, 设定新的资料
32     memcpy( ts_DA.NowBox , po , sizeof( po ));
33 }

```

### 程序说明

- (1) 第 4~10 行：根据上表列出旋转索引数组。
- (2) 第 18~21 行：利用索引数组取得旋转后方块的位置。
- (3) 第 23~29 行：判定新的方块位置是否超出游戏框或该位置是否有方块，四个格子中只要有一个不是空位，就不旋转它。
- (4) 第 32 行：记录新的方块位置。

在这个俄罗斯方块的范例中，已经介绍了建立大型游戏的基本架构，包括游戏框架、玩家操作、游戏判断、绘制，以及如何利用 C++ 继承的方式来设定游戏处理方式。如果再花点心思，加入音乐音效或闪光特效，或是加些选单、排行榜，或是将方块改成魔法宝石或魔法气泡等等，就可以将游戏设计得更出色。

## 12.2 抢娃娃游戏

在一座遥远的深山里有一个小女孩，她向七彩琉璃珠许下了埋藏在心里已久的愿望。她并非要求金钱、权力、爱情，她要的只是一个单纯的小女孩的童年。于是上天应许她的要求，从天上降下许许多多的小娃娃。

这是一个抢娃娃的游戏，玩家要扮演小女孩。在有限的时间内娃娃由天上一个个掉下来，玩家就要操作主角快速地把娃娃捡起来。

这个范例 ch12\_treasure（见附书光盘）示范了一个平面游戏概念，将延续“俄罗斯方块”的架构，深入研究 2D 动作游戏中应有的组件。

### 1. 游戏介绍

运行程序后会出现主标题，按【Z】键即可开始游戏，如图 12-8 所示。

在 60 秒内，娃娃会一个个掉下来，每秒掉 1 个，最多 32 个，如图 12-9 所示。



图 12-8



图 12-9

玩家按方向键即可移动去捡娃娃。该游戏提供 3 个力量，可以加快主角移动速度，每用一次可维持 5 秒。画面如图 12-10 所示。



图 12-10

捡一个娃娃可得 100 分并提供 2 秒的连续时间，并将其等分为 4 个单位，得分为连续单位数的平方 $\times 100$ 。

时间结束后游戏结束，可再按【Z】键重新挑战。画面如图 12-11 所示。



图 12-11

## 2. 坐标系

在计算机屏幕上画点与线的时候，通常是以行与列的方式来标明位置，所以大多以左上角作为屏幕坐标的原点，并作为测量与定位的参考。例如，在屏幕的(100, 200)位置上贴一张大小为 50 $\times$ 50 的位图，则(100, 200)即是它的贴图坐标，如图 12-12 所示。

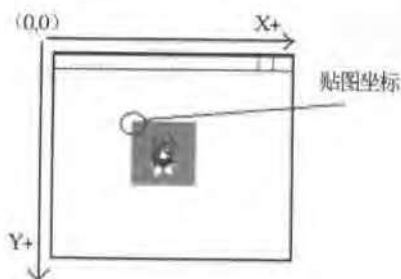


图 12-12

游戏设计者可能设计了一个很大的游戏世界，如城堡、大草原、森林等，以提供玩家冒险。游戏世界的坐标系统通常和屏幕坐标系一致以方便管理，以世界中左上角(0,0)为基准，称为世界坐标系(World)。游戏世界中的对象都是以世界原点为基准所产生的，一般只需要代表位置的 X, Y 二维即可，也加上高度 Top，让对象飞起来。世界坐标的画面如图 12-13 所示。

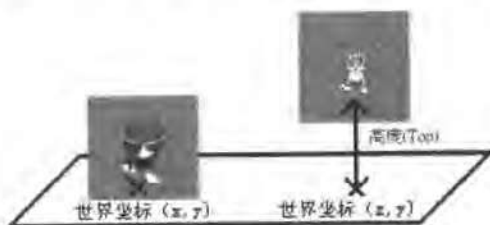


图 12-13

玩家的冒险世界可能非常庞大，超过屏幕大小。如果将世界缩小又显现不出气势，我们需要一组坐标将世界中某个位置转换到屏幕坐标的原点，这个转换坐标称为世界滚动条(Screen)，公式为：

$$\text{屏幕坐标} = \text{世界坐标} - \text{世界滚动条}$$

形象的演示如图 12-14 所示。

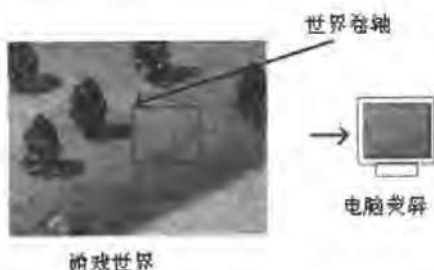


图 12-14

游戏世界中的任何对象可能是在地面上站着、在天上飞或在水中游，自然不能将该坐标直接当成贴图坐标来用。以图片左上角为原点，取一点当成基准点(Origin)，使其对应到该对象的世界中。该对应关系如图 12-15 所示。

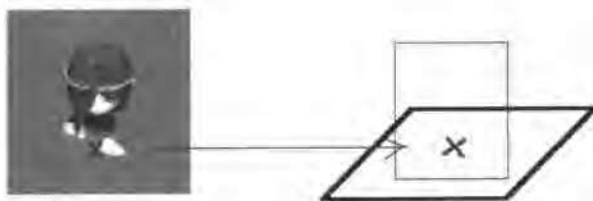


图 12-15

最后，世界坐标转为贴图坐标的公式如下：

$$\text{贴图坐标}(x) = \text{世界坐标}(x) - \text{滚动条坐标}(x) - \text{基准点}(x)$$

$$\text{贴图坐标}(y) = \text{世界坐标}(y) - \text{滚动条坐标}(y) - \text{基准点}(y) - \text{高度}(top)$$

形象的演示如图 12-16 所示。

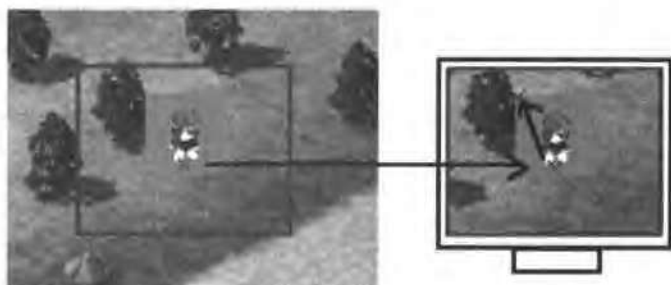


图 12-16

### 3. 材质串制作

捡娃娃所用的图案使用各个单张的图素，并将其包装成一个动作行为。游戏进行时一次只贴一小格，而不用再去计算材质的切割区块。如图 12-17 所示。

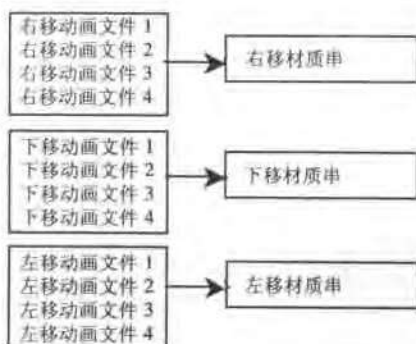


图 12-17

为了方便管理编写对象 d3dCartoon，可以在 myd3d.h 中看到以下声明：

```

1  class d3dCartoon
2  {
3  private :
4      int m_Num ;           //总数量
5      int m_Gx , m_Gy ;     //2D 分页
6      int m_Width , m_Height ;
7      d3dTexture * m_Texture ;
8  private :
9      void Create( int num );
10 public :
11     int m_Ox , m_Oy ;      //基准点
12 public :                  //建立材质
13     int CreateID( LPCTSTR strFormat , int from , DWORD ColorKey );
14     int Create2D( LPCTSTR strFormat , DWORD ColorKey );
15 public :                  //取得材质
16     d3dTexture* getTexture( int index );
17     d3dTexture* getTexture( int gx , int gy );

```

```

18 public :                               //取得图片资料
19     inline int getNum(){ return m_Num ; }
20     inline int getWidth(){ return m_Width ;}
21     inline int getHeight(){ return m_Height ;}
22 public :                               //地图绘制
23     void Draw2D( int sx , int sy , int sw , int sh );
24 public :
25     d3dCartoon();
26     ~d3dCartoon();
27     void Init();
28     void Release();
29 };

```

这里可以看到除了记录材质指针（d3dTexture）数量、贴图基准点（m\_Ox,m\_Oy）外，还可管理线性材质串与平面材质串，线性材质串指的是很简单的动作串，播完后可回放、停止或换动作。而平面材质大多用在大地图。由于系统先天的限制，有时超大型的图文件不一定可以成功加载到内存中，贴图时大多只局限于屏幕窗口大小，全部加载就显得过于浪费。于是，我们先将超大型图文件切割成多张相同长宽小图，游戏进行中若需要显示时再加载即可。如图 12-18 所示。

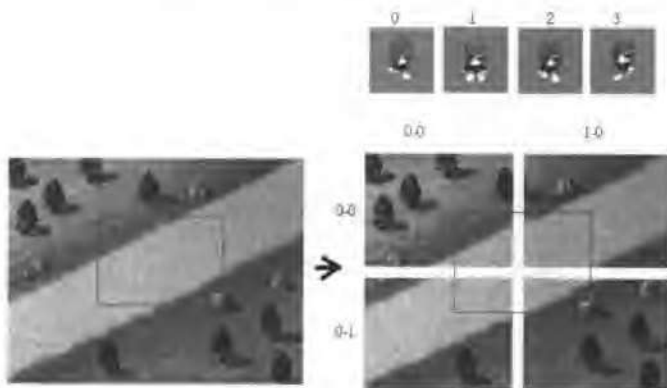


图 12-18

先来看看线性材质串是如何读入的，可以在 myd3d.cpp 找到下面程序代码。

```

1  int d3dCartoon::CreateID( LPCTSTR strFormat , int from , DWORD ColorKey )
2  {
3      int i , num ;
4      char ch[256] ;
5      //判断共有多少动画格
6      num = from ;
7      while( true )
8      {
9          wsprintf( ch , strFormat , num );
10         if( !mylibFileIs( ch ))
11             break ;
12         num ++ ;
13     }
14     if( num == from )return 0 ;
15     //建立材质

```

```
16     Create( num );
17     //加载材质
18     for( i = from ; i < num ; i++ )
19     {
20         wsprintf( ch , strFormat , i );
21         m_Texture[i].Create( ch , ColorKey );
22     }
23     //记录单元格长宽
24     m_Width = m_Texture[0].getWidth();
25     m_Height = m_Texture[0].getHeight();
26     //结束
27     return m_Num ;
28 }
```

#### 程序说明

(1) 第 6~14 行: 将加载格式化的文字转成文件名, 寻找是否有该文件名, 直到没有文件为止, 以计算共有多少实际存在的图素。

(2) 第 16 行: 由前面得到的数量配置内存。

(3) 第 18~22 行: 将格式化文字再转成文件名称, 一个个加载到内存中。

这个方式的优点是不用考虑图素的数量, 只要名称有一定的格式, 随时都可以增加、插入或删除图素, 只要注意图素的档名是否有连续就可以了。

(4) 第 24~25 行: 记录图素单元格的长宽。

平面材质串的做法和线性材质串的差不多, 只是多了长宽格数与总长度。

```
1  int d3dCartoon::Create2D( LPCTSTR strFormat , DWORD ColorKey )
2  {
3      int i , j , num ;
4      int gx , gy ;
5      char ch[256] ;
6      //取得纵向张数
7      gx = 0 ;
8      while( 1 )
9      {
10         wsprintf( ch , strFormat , gx , 0 );
11         if( !mylibFileIs( ch ))
12             break ;
13         gx ++ ;
14     }
15     //取得直向张数
16     gy = 0 ;
17     while( 1 )
18     {
19         wsprintf( ch , strFormat , 0 , gy );
20         if( !mylibFileIs( ch ))
21             break ;
22         gy ++ ;
23     }
24     //计算总张数
```



```

25     num = gx * gy ;
26     if( num == 0 )return 0 ;
27     //建立材质
28     Create( num );
29     //加载材质
30     for( j = 0 ; j < gy ; j++ )
31     for( i = 0 ; i < gx ; i++ )
32     {
33         wsprintf( ch , strFormat , i , j );
34         m_Texture[ j * gx + i ].Create( ch , ColorKey );
35     }
36     //取得总长宽
37     for( i = 0 ; i < gx ; i++ )
38         m_Width += m_Texture[i].getWidth();
39     for( j = 0 ; j < gy ; j++ )
40         m_Height += m_Texture[j*gx].getHeight();
41     //结束，并记录资料
42     m_Gx = gx ;
43     m_Gy = gy ;
44     return m_Num ;
45 }

```

### 程序说明

- (1) 第 7~14 行：计算横向图素数量。
- (2) 第 16~23 行：计算纵向图素数量。
- (3) 第 25~28 行：横纵向数量相乘即可得总数量，并建立材质。
- (4) 第 30~35 行：加载材质。
- (5) 第 37~40 行：取得平面总长宽。

线性材质串与平面材质串最大的不同点为绘制部分，前者只要贴一张，并只要取得索引图素后贴上就可以了；后者则需要画一个平面，可能是整张材质中的小区块，或是由许多图素各切一个部分组合而成。二者的不同之处如图 12-19 所示。

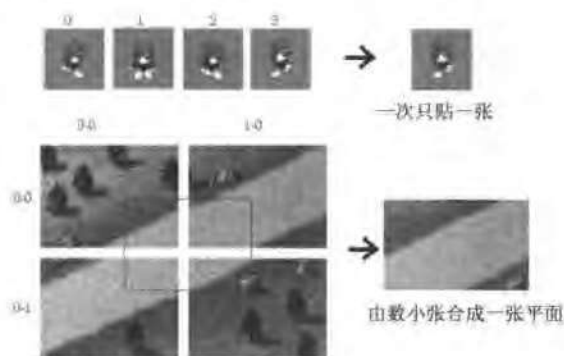


图 12-19

下面的程序代码就是示范如何将分割的图素拼成一张地图的。

```

1 void d3dCartoon::Draw2D( int sx , int sy , int sw , int sh )
2 {

```

```

3      int i , j ;
4      int x , y ;
5      int w , h ;
6      d3dTexture *texture ;
7      //判断是否已加载
8      if( m_Texture == NULL )
9          return ;
10     //取得材质第一张大小
11     w = m_Texture[0].getWidth();
12     h = m_Texture[0].getHeight();
13     //计算左上角图格
14     for( j = 0 , y = -sy ; j < m_Gy ; j++ , y += h )
15     for( i = 0 , x = -sx ; i < m_Gx ; i++ , x += w )
16     {
17         //判定是否在屏幕内
18         if(( x < sw ) && ( y < sh ))
19             if((( x + w ) > 0 ) && (( y + h ) > 0 ))
20             {
21                 //取得材质
22                 texture = getTexture( i , j );
23                 if( !texture )break ;
24                 //绘制
25                 texture->BlitFast( x , y );
26             }
27     }
28 }

```

#### 程序说明

(1) 第11~12行：取得单张图素大小。

(2) 第14~27行：由左上角开始，将世界滚动条坐标转成屏幕坐标，并判断左上右下四个点是否在屏幕显示区内，如果是，即进行贴图。这里所谓的世界滚动条坐标，指的是该坐标的贴图位置对应于屏幕左上角(0,0)的位置。将世界滚动条坐标除以单张图素长宽，商数即为地图格，而余数就是该格对应到屏幕原点。

在绘制之前，将每一张地图格转成屏幕坐标后，再判断是否有超出屏幕范围的。例如，我们将地图切成4×3格，共12块，假设世界坐标座落于格位1-0的位置，这时我们只要贴1-0、2-0、1-1、2-1四张即可，如图12-20所示。

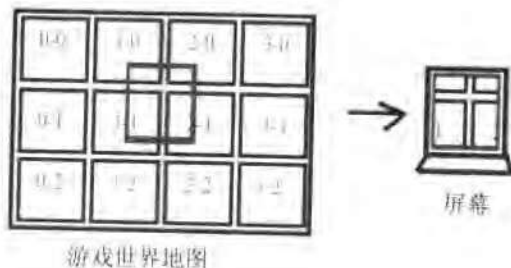


图 12-20

## 4. 排序贴图

站在某个场景中向一个方向看去，你就会发现有的对象在前，有的在后。一般来说，前面的对象会遮住后面的对象，由此人们可以判断对象的远近。在 3D 场景中，大多是用 Z 缓冲区 (ZBuffer) 来判定模型的远近，而 2D 游戏通常是采用“画家算法” (Painter Algorithm)，即远距离的对象先画，画近距离时就会盖住远距离对象。排序贴图就是以对象世界坐标的 Y 轴当成深度，判定该对象的远近。排序贴图的具体例子如图 12-21 所示。

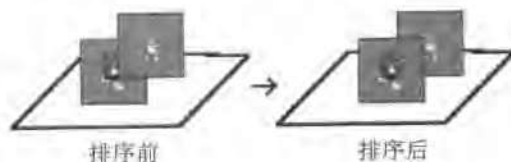


图 12-21

2D 游戏不代表不用三维坐标，有些场景可能会出现飞机、鸟或掉落的岩石等飞行物体，仍是用世界坐标 Y 轴作为深度进行排序，但实际贴图的 Y 轴位置还要再加上高度距离，视觉上就少了立体感。这时，除自身的对象图外，可能还需要影子，如图 12-22 所示。



图 12-22

有了以上的概念后，接下来讨论程序是如何进行排序的。和游戏循环框架一样，利用对象继承的观念，制作一个用来管理游戏里所有对象位置及排序的框架，而游戏中的对象只要继承这个框架就可以了。我们可以在 appProc.h 中看到这个框架的内容。

```

1  class gameObj
2  {
3  public:          //排序串行
4      gameObj * m_Exit;    //串行下一个
5      gameObj * m_Next;    //串行上一个
6  public:
7      float m_x, m_y, m_top;
8  public:
9      virtual void SortDraw();
10     void AddSort();
11 };
  
```

### 程序说明

- (1) 第 4~6 行：用来串接对象的指针。
- (2) 第 7 行：对象的世界坐标及离地面的高度 (m\_top)。
- (3) 第 9 行：排序后对象调用绘图的虚拟函数。

(4) 第10行：加入排序串行函数。

这里用数据结构中的串行 (serial-access memory) 进行排序。假设有 ABCD 四个对象，且世界坐标中  $A_y < B_y < C_y < D_y$ 。对 A 而言，它的上面即是 B，所以 A 的上接指针 (m\_Exit) 即指向 B；对 B 而言，B 的上接指针即为 C；一直到最后一个对象时，已经没有任何对象在它的上面，所以上接指针为 NULL (空)。反过来说，由于 D 是盖在 C 上面的，所以 D 的下接指针 (m\_Next) 即为 C，也同样的覆盖一个，最后一个的下接指针也是 NULL。A、B、C、D 的关系如图 12-23 所示。

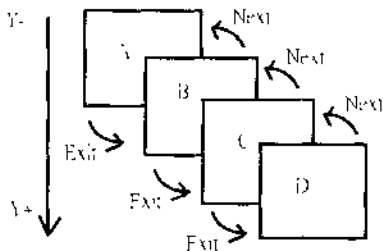


图 12-23

另外还需要一个用来记录串行中最底层的“根”，可以在 appProc.cpp 中看到排序串行的对象指针声明。

```
gameObj * app_Sort ;           //排序用串行基类
```

了解了串行的原则后，可以写出串接的函数，如下所示。

```
1 void gameObj::AddSort()
2 {
3     gameObj *sort ;
4     gameObj *exit ;
5     m_Exit = NULL ;
6     m_Next = NULL ;
7     //若基类没有对象，就直接加入
8     sort = app_Sort ;
9     if( !sort )
10    {
11        app_Sort = this ;
12        return ;
13    }
14    //Y若为最小，加入第一个
15    if( m_y <= sort->m_y )
16    {
17        m_Exit = sort ;
18        sort->m_Next = this ;
19        app_Sort = this ;
20        return ;
21    }
22    //开始依序判断 z
23    while( sort )
24    {
25        if( m_y <= sort->m_y )
```

```

26         {
27             //加入下一个
28             if( sort->m_Next )
29                 sort->m_Next->m_Exit = this ;
30             m_Next = sort->m_Next ;
31             //加入上一个
32             sort->m_Next = this ;
33             m_Exit = sort ;
34             return ;
35         }
36         exit = sort ;
37         sort = sort->m_Exit ;
38     }
39     //Y为最大值, 加入到最后一个
40     exit->m_Exit = this ;
41     m_Next = exit ;
42 }

```

## 程序说明

- (1) 第 5~6 行：将对象本身的串接指针清空。
  - (2) 第 8~13 行：如果串行的根为空，直接将它加入。
  - (3) 第 15~21 行：如果该对象中的深度比最底层的还要远，则表示原来的根盖在它上面，串接后将把它当成根。
  - (4) 第 23~38 行：由深度最远的开始，依次判定该插入哪个对象之中。
  - (5) 第 40~41 行：该对象的深度比任何对象都近，就接在最后一个对象上。
- 对象在加入排序后，由根开始深度由远到近地串连起来。在绘制时同样也由根开始一个一个地贴，如下所示。

```

1 void appSortDraw()
2 {
3     gameObj * sort ;
4     sort = app_Sort ;
5     while( sort )
6     {
7         sort->SortDraw();
8         sort = sort->m_Exit ;
9     }
10 }

```

## 程序说明

- (1) 第 4 行：取出对象的根。
- (2) 第 7 行：调用虚拟函数来绘制该对象。
- (3) 第 8 行：取得盖在上面的对象。

使用串行的优点在于不用考虑对象的数量，可以随意的增减。但是，如果数量较多，间接影响到游戏性能，最好事先判断是否可以正确的贴在屏幕上再进入排序。

最后要注意的是，在串行的全部清除之前，同一个对象只可以进入排序一次，否则会因串行指针重叠而造成无限循环。

## 5. 游戏对象设计

接下来,以对象 gameObj 为基类,将游戏中的元素、动画包装成可在游戏中使用的对象,可以根据游戏需要定义为各类不同的游戏对象。下面即为对四处移动的主角及掉落的娃娃对象的声明。

```

1  class gameRole : public gameObj
2  {
3  private :
4      int m_GridTime , m_GridNextTime ; //播放时间
5      int m_Index ;                      //目前格位索引
6      int m_News ;                       //方向
7  public :
8      d3dCartoon *m_Cartoon ;            //动画串
9      int m_Alpha ;                      //对象颜色
10 public :
11     void SetCartoon( d3dCartoon * cartoon , int NextTime , int News );
12     void Draw();
13     void SortDraw();
14 };

```

### 程序说明

(1) 第4~6行: 播放动画串的计时工具。

(2) 第8行: 记录目前所使用材质串, 并且判定该对象是否正在使用。

(3) 第9行: 游戏中主角取得娃娃时, 娃娃以淡出的方式显示, m\_Alpha 用来计算淡出的混色值。

对象 gameRole 绘制的工作分成二次, 第一次由游戏循环调用 gameRole::Draw 函数, 以对象的世界坐标为中心画上影子, 并且进入排序。gameRole::Draw 函数如下所示。

```

1  void gameRole::Draw()
2  {
3      int x , y ;
4      //计算下一格
5      m_GridTime -= di_DDD._time ;
6      while( m_GridTime < 0 )
7      {
8          m_GridTime += m_GridNextTime ;
9          m_Index ++ ;
10         if( m_Index >= m_Cartoon->getNum() )
11             m_Index = 0 ;
12     }
13     //画影子
14     x = (int)m_x - gm_Da.mScreenX ;
15     y = (int)m_y - gm_Da.mScreenY ;
16     d3d_Device->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_ZERO );
17     d3d_Device->SetRenderState( D3DRS_DESTBLEND , D3DBLEND_INVSRCOLOR );
18     d3d_Device->SetTextureStageState( 0 , D3DTSS_ALPHAOP , D3DTOP_MODULATE );
19     if( m_Alpha )
20         app_Shadow.BltFast( x - 32 , y - 32 , x + 32 , y + 24 , D3DCOLOR_ARGB( m_Alpha ,
            m_Alpha , m_Alpha , m_Alpha ));

```

```

21     else
22         app_Shadow.BltFast( x - 32 , y - 32 , x + 32 , y + 24 , -1 );
23         //加入排序串行
24         x -= m_Cartoon->m_Ox ;
25         y -= m_Cartoon->m_Oy - m_top ;
26         if(( x >= 640 ) || ( y >= 480 ))
27             return ;
28         if((( x + m_Cartoon->getWidth() ) < 0 ) || (( y + m_Cartoon->getHeight() ) < 0 ))
29             return ;
30         AddSort();
31     }

```

## 程序说明

- (1) 第 5~12 行：计算播放格是否该往下移。
- (2) 第 14~22 行：将世界坐标转换成屏幕坐标，以此为中心贴上影子。
- (3) 第 24~30 行：将屏幕坐标转换成贴图坐标，计算是否在显示区内，如果是，则将该对象加入到贴图串行中。

第二次绘图调用是在贴图排序后。当所有组件的贴图顺序都决定好后，即由虚拟函数响应所调用的 `gameRole::SortDraw` 函数。

```

1  void gameRole::SortDraw()
2  {
3      int dx , dy ;
4      d3dTexture *texture ;
5      //判定是否有对象
6      if( !m_Cartoon )return ;
7      //计算绘制位置
8      dx = (int) m_x - m_Cartoon->m_Ox - gm_Da.mScreenX ;
9      dy = (int)( m_y - m_top )- m_Cartoon->m_Oy - gm_Da.mScreenY ;
10     //取得并绘制材质
11     d3d_Device->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_SRCALPHA );
12     d3d_Device->SetRenderState( D3DRS_DESTBLEND , D3DBLEND_INVSRCALPHA );
13     d3d_Device->SetTextureStageState( 0 , D3DTSS_ALPHAOP , D3DTOP_MODULATE );
14     if( texture = m_Cartoon->getTexture( m_Index ))
15         if( m_Alpha )
16             texture->BltFast( dx , dy , dx + texture->getWinth() , dy + texture->
                getHeight() , D3DCOLOR_ARGB( m_Alpha , 255 , 255 , 255 ));
17         else
18             texture->BltFast( dx , dy );
19 }

```

## 程序说明

- (1) 第 8~9 行，将世界坐标转换成贴图坐标。
- (2) 第 11~17 行，由动画格索引取得将绘制的图素，并贴到屏幕上。

## 6. 游戏场景初始化

在进入“接娃娃”游戏前，先来看看为游戏定义了什么东西。

```

1  d3dCartoon app_Back ;           //背景
2  d3dCartoon app_Role[4] ;        //娃娃
3  d3dCartoon app_Prop[3] ;        //掉落娃娃
4  gameObj * app_Sort ;            //排序用串行基类
5  d3dTexture app_Shadow ;         //影子
6  d3dTexture app_Title ;         //标题
7  gameRole app_GamePlay ;         //玩家信息
8  gameRole app_GameProp[32] ;     //掉落对象
9  typedef struct _GAME_DATA
10 {
11     //移动信息
12     int mWidth , mHeight ;
13     int mScreenX , mScreenY ;
14     //下一个掉落物品时间
15     int rTimeDown ;
16     //得分
17     int sTimeGame ;              //游戏时间
18     int sTimeGet ;              //物品取得时间
19     int sScore ;                //得分
20     int sNum ;                  //取得数量
21     //power
22     int sTimePower ;            //力量时间
23     int sPowerNum ;            //数量
24 }GAME_DATA ;
25 GAME_DATA gm_Da ;

```

和俄罗斯方块一样,在进入游戏之前先将所有准备好的图片加载到内存,函数 `appLoadGameData()` 即用于处理这份工作。

```

1  BOOL appLoadGameData()
2  {
3      int i ;
4      //加载背景
5      app_Back.Create2D( "m%2.2d_%2.2d.bmp" , 0 );
6      //加载影子
7      app_Shadow.Create( "影子02.bmp" , 0 );
8      //加载掉落物品
9      app_Prop[0].Create1D( "松鼠%2.2d.bmp" , 0 , D3DCOLOR_ARGB( 255 , 0 , 255 , 0 ));
10     app_Prop[1].Create1D( "恶魔%2.2d.bmp" , 0 , D3DCOLOR_ARGB( 255 , 0 , 255 , 0 ));
11     app_Prop[2].Create1D( "魔鹿%2.2d.bmp" , 0 , D3DCOLOR_ARGB( 255 , 0 , 255 , 0 ));
12     //设定掉落物品基准点
13     for( i = 0 ; i < 3 ; i++ )
14     {
15         app_Prop[i].m_Ox = 70 ;
16         app_Prop[i].m_Oy = 100 ;
17     }
18     //加载娃娃
19     app_Role[0].Create1D( "巴冷_01_%2.2d.bmp" , 0 , D3DCOLOR_ARGB( 255 , 0 , 255 , 0 ));
20     app_Role[1].Create1D( "巴冷_03_%2.2d.bmp" , 0 , D3DCOLOR_ARGB( 255 , 0 , 255 , 0 ));

```



```

21     app_Role[2].CreateID( "巴冷_05_&2.2d.bmp" , 0 , D3DCOLOR_ARGB( 255 , 0 , 255 , 0 ) );
22     app_Role[3].CreateID( "巴冷_07_&2.2d.bmp" , 0 , D3DCOLOR_ARGB( 255 , 0 , 255 , 0 ) );
23     //设定娃娃基准点
24     for( i = 0 ; i < 4 ; i++ )
25     {
26         app_Role[i].m_Ox = 70 ;
27         app_Role[i].m_Oy = 100 ;
28     }
29     //初始化第一个娃娃
30     app_GamePlay.SetCartoon( app_Role , 100 , 1 );
31     app_GamePlay.m_x = 320.0f ;
32     app_GamePlay.m_y = 240.0f ;
33     //其他设定
34     d3d_Device->SetSamplerState( 0 , D3DSAMP_MAGFILTER , D3DTEXF_POINT );
35     d3d_Device->SetSamplerState( 0 , D3DSAMP_MINFILTER , D3DTEXF_POINT );
36     d3d_Device->SetRenderState( D3DRS_ZENABLE , false );
37     return true ;
38 }

```

### 程序说明

- (1) 第5行：将四张 640×480 的切割图加载内存，组成一个 1280×960 的游戏场景。
- (2) 第9~17行：将落下的娃娃图素加载内存，并设定贴图基准点。
- (3) 第19~28行：将四个方向的主角图素加载到内存，并设定贴图基准点。
- (4) 第29~32行：将面向下方的图素设定给主角，并将主角的世界坐标定义于 (320,240)。

## 7. 游戏绘制

终于可以看到游戏世界显示在屏幕上的样子，下面是绘图函数 appTreasure::Render()。

```

1     void appTreasure::Render()
2     {
3         d3dTexture d3dRt ;
4         //清空
5         d3dClear();
6         //排序串行清空
7         appSortInit();
8         //绘制
9         d3d_Device->BeginScene();
10        d3d_Device->SetRenderState( D3DRS_SRCBLEND , D3DBLEND_SRCALPHA );
11        d3d_Device->SetRenderState( D3DRS_DESTBLEND , D3DBLEND_INVSRCALPHA );
12        d3d_Device->SetRenderState( D3DRS_ALPHABLENDENABLE , true );
13        d3d_Device->SetTextureStageState( 0 , D3DTSS_ALPHAOP , D3DTOP_SELECTARG2 );
14        //画背景
15        app_Back.Draw2D( gm_Da.mScreenX , gm_Da.mScreenY , 640 , 480 );
16        //画信息框
17        d3dRt.BlitFast( 15 , 15 , 200 , 125 , 0x8F000000 );
18        //画娃娃影子，并加入排序串行
19        d3d_Device->SetTextureStageState( 0 , D3DTSS_ALPHAOP , D3DTOP_SELECTARG1 );
20        app_GamePlay.Draw();

```

```

21      //画掉落物品
22      DrawProp();
23      //画排序串行
24      appSortDraw();
25      d3d_Device->EndScene();
26      //输出信息文字
27      DrawText();
28  }

```

#### 程序说明

- (1) 第7行：清空排序对象的根。
- (2) 第9~17行：绘制大地图及游戏信息框。
- (3) 第19~22行：绘制主角及娃娃的影子，并加入贴图排序。
- (4) 第24~25行：将排序好的对象进行对象的第二次绘制。

### 8. 主角移动

在主角的移动函数里，除了要响应玩家所操作的动作外，还要将主角限定在游戏场景内，并计算世界滚动条的位置，使得主角的位置对准屏幕中心。

```

1  void appTreasure::Move()
2  {
3      float speed ;
4
5      //游戏经过的时间
6      gm_Da.sTimeGame -= di_DDD._time ;
7      gm_Da.sTimeGet  -= di_DDD._time ;
8      gm_Da.sTimePower -= di_DDD._time ;
9      //判定游戏是否结束
10     if( gm_Da.sTimeGame <= 0 )
11     {
12         appTreasureGameOverRun();
13         return ;
14     }
15     //使用加速
16     if( di_DDD.Bn[0] )
17     if( gm_Da.sTimePower <= 0 )
18     if( gm_Da.sPowerNum > 0 )
19     {
20         gm_Da.sPowerNum -- ;
21         gm_Da.sTimePower = 5000 ;
22     }
23     //娃娃移动速度
24     if( gm_Da.sTimePower > 0 )
25         speed = ((float)di_DDD._time )* 0.40f ;
26     else
27         speed = ((float)di_DDD._time )* 0.20f ;
28     //娃娃移动方向
29     switch( di_DDD.News4ing )

```

```

30     {
31         case 1 ://下
32             app_GamePlay.m_y += speed ;
33             app_GamePlay.SetCartoon( &app_Role[0] , 100 , 1 );
34             break ;
35         case 5 ://上
36             app_GamePlay.m_y -= speed ;
37             app_GamePlay.SetCartoon( &app_Role[2] , 100 , 5 );
38             break ;
39         case 3 ://右
40             app_GamePlay.m_x += speed ;
41             app_GamePlay.SetCartoon( &app_Role[1] , 100 , 3 );
42             break ;
43         case 7 ://左
44             app_GamePlay.m_x -= speed ;
45             app_GamePlay.SetCartoon( &app_Role[3] , 100 , 7 );
46             break ;
47     }
48     //移动修正
49     if( app_GamePlay.m_x < 0 )
50         app_GamePlay.m_x = 0 ;
51     else if( app_GamePlay.m_x > app_Back.getWidth() )
52         app_GamePlay.m_x = (float)app_Back.getWidth() ;
53     if( app_GamePlay.m_y < 0 )
54         app_GamePlay.m_y = 0 ;
55     else if( app_GamePlay.m_y > app_Back.getHeight() )
56         app_GamePlay.m_y = (float)app_Back.getHeight() ;
57     //滚动条修正
58     gm_Da.mScreenX = (int)app_GamePlay.m_x - 320 ;
59     gm_Da.mScreenY = (int)app_GamePlay.m_y - 240 ;
60     if( gm_Da.mScreenX < 0 )
61         gm_Da.mScreenX = 0 ;
62     else if( gm_Da.mScreenX > gm_Da.mWidth )
63         gm_Da.mScreenX = gm_Da.mWidth ;
64     if( gm_Da.mScreenY < 0 )
65         gm_Da.mScreenY = 0 ;
66     else if( gm_Da.mScreenY > gm_Da.mHeight )
67         gm_Da.mScreenY = gm_Da.mHeight ;
68 }

```

## 程序说明

- (1) 第 6~14 行：计算更新游戏定时器，当游戏时间超过一分钟时游戏结束。
- (2) 第 16~22 行：判定玩家是否使用特殊能力，如果有，就将加速定时器定 5 秒，让主角可以在这段时间内加快移动速度。
- (3) 第 24~27 行：计算玩家的移动速度，公式为：距离=速率×时间，如果加速定时器未超过 5 秒，速率就是 0.4，否则为 0.2。
- (4) 第 29~47 行：按玩家的操作方向移动主角的世界坐标，并设定该方向的材质串。
- (5) 第 49~57 行：修正主角的移动位置，将主角的世界坐标锁定在游戏场景内。

(6) 第 58~67 行: 调整世界滚动条坐标, 利用主角的世界坐标向左上移位, 以便将主角的世界坐标对准屏幕中心点。

除了随玩家控制而四处跑的主角外, 掉落的娃娃也须依序加入, 函数 `appTreasure::AddObj()` 的作用为每经过 1 秒, 就随机选个地点, 让娃娃加入场景中。

```
1      void appTreasure::AddObj()
2      {
3          int i ;
4          //掉落定时器
5          gm_Da.rTimeDown -= di_DDD._time ;
6          if( gm_Da.rTimeDown > 0 )return ;
7          gm_Da.rTimeDown += 1000 ;
8          //将娃娃加入场景
9          for( i = 0 ; i < 32 ; i++ )
10             if( !app_GameProp[i].m_Cartoon )
11             {
12                 app_GameProp[i].SetCartoon( &app_Prop[rand()%3] , 100 , -1 );
13                 app_GameProp[i].m_x = (float)( rand()%( app_Back.getWidth() - 200 )
14                     + 100 );
15                 app_GameProp[i].m_y = (float)( rand()%( app_Back.getHeight() - 200 )
16                     + 100 );
17                 app_GameProp[i].m_top = 480.0f ;
18                 app_GameProp[i].m_Alpha = 0 ;
19                 return ;
20             }
21     }
```

#### 程序说明

(1) 第 9~10 行: 用材质串 `m_Cartoon` 来判定该内存是否正在运行。如果找到空的, 就将该娃娃加入到场景中。

(2) 第 12 行: 从 3 组娃娃材质串中随机取一组给娃娃对象。

(3) 第 13~15 行: 设定娃娃的世界坐标及高度。

函数 `appTreasure::GetProp()` 用来计算主角与娃娃是否碰撞在一起, 并计算得分。下面来看看它的运作过程。

```
1
2      void appTreasure::GetProp()
3      {
4          int i ;
5          int x , y ;
6          int Score ;
7          float len ;
8          for( i = 0 ; i < 32 ; i++ )
9              if( app_GameProp[i].m_Cartoon )
10                 if( app_GameProp[i].m_top <= 0.0f )
11                     if( app_GameProp[i].m_Alpha )
12                     {
13                         app_GameProp[i].m_Alpha -= di_DDD._time ;
```

```

14         if( app_GameProp[i].m_Alpha <= 0 )
15             app_GameProp[i].m_Cartoon = NULL ;
16     }else
17     {
18         //取两者距离
19         x = abs((int)( app_GamePlay.m_x - app_GameProp[i].m_x ));
20         y = abs((int)( app_GamePlay.m_y - app_GameProp[i].m_y ));
21         len = sqrtf((float)( x * x + y * y ));
22         if( len < 32.0f )
23         {
24             //透明值索引
25             app_GameProp[i].m_Alpha = 255 ;
26             //计算分数
27             if( gm_Da.sTimeGet > 0 )
28                 Score = gm_Da.sTimeGet / 500 + 1 ;
29             else Score = 1 ;
30             gm_Da.sScore += ( Score * Score * 100 ) ;
31             gm_Da.sNum ++ ;
32             gm_Da.sTimeGet = 2000 ;
33         }
34     }
35 }
  
```

## 程序说明

(1) 第 9~10 行：判断数组中的娃娃是否已落到地面。

(2) 第 11~15 行：捡了娃娃之后，将该物品的 `m_Alpha` 值设为 255，用来作为娃娃淡出的混色值（Alpha），这里随时间让它越来越淡。当完全透明时（`m_Alpha<=0`），将娃娃贴图清除，以便重新使用。

(3) 第 19~21 行：利用毕式定理（斜边的平方等于直角两边的平方和）求主角与娃娃两者世界坐标的距离。

(4) 第 22~33 行：当主角与娃娃两者距离小于 32 时，判定两者已碰撞，并计算得分。这里并不直接清除它，仍将娃娃慢慢淡出屏幕，所以将 `m_Alpha` 值设为 255，再由 11~15 行来完成。

到此为止，已经将 2D 游戏世界架构讲解完毕，其实不论是动作游戏、角色扮演或实时战略，内容多少都有些差异，但整个场景建立绝对离不开本章所介绍的内容，希望您能完全理解它们并制作出更加好玩、刺激的游戏来。

# 附录 A DirectDraw 制作游戏秘籍大公开

Direct Graphics 如前所述，完全可以取代原有的 DirectDraw，并且可以很轻松的完成各种效果。Direct Graphics 完全建立在 3D 基础上，这个架构主要跟随硬件产生，可充分使用硬件所带来的视觉效果。当玩家硬件并不如想象中强大时，也就成了 Direct Graphics 最大的败笔。大到全 3D 的第一人称射击，小到简单的 2D 的小蜜蜂，当硬件不支持或是显示卡驱动程序未更新，又或者部分硬件加速功能损坏，都会直接影响游戏的效能，游戏的娱乐性自然也会大打折扣。

随着硬件加速功能的增强，DirectDraw 会逐步退出游戏市场。从另一角度来看，如果设计的只是 2D 的小游戏，在不考虑硬件情况下，又期待能达到非常好的游戏性能时，DirectDraw 就比 Direct Graphics 实用多了，因为 DirectDraw 本身就是专为 2D 游戏所开发的组件。

本附录将介绍把 DirectDraw 作为主要绘图引擎，配合 DirectX 其他组件，制作出几个简单的 2D 游戏程序的范例。

## A.1 程序中的各个自定义函数

这些游戏范例程序中运用了 DirectX 7.0 设计主要的功能。在这一节里，先来看看各个范例中所用到的共享自定义函数，这些函数的功能主要是初始化各种 DirectX 对象以及建立缓冲区。

### A.1.1 初始化与建立 DirectX 对象

在本附录后面的每一个范例中，都使用了 DirectX 来制作全屏幕的游戏，包含 DirectDraw、DirectSound、DirectInput，而 InitDD()、InitDS()、InitDI()这三个自定义函数则是用来初始化这些 DirectX 对象。

#### 1. 建立 DirectDraw

初始化与建立 DirectDraw 的函数为 InitDD()，其变量声明与程序代码内容如下所示。

##### 程序代码

```
1 //声明 DirectDraw 变量
2 LPDIRECTDRAW7 DD;
3 LPDIRECTDRAWSURFACE7 DDSur;
4 LPDIRECTDRAWSURFACE7 DDBuf;
5 LPDIRECTDRAWSURFACE7 DDPla[19];
6 DDSCAPS2 DDcaps;
7 DDSURFACEDESC2 DDde;
8 HRESULT result;
9 DDCOLORKEY key;
10 void canvasFrame::InitDD()
11 {
```

```

12 //建立 DirectDraw 对象
13 result = DirectDrawCreateEx(NULL, (VOID**)&DD,
14                               IID_IDirectDraw7, NULL);
15 if (result != DD_OK)
16 {
17     MessageBox("建立 DirectDraw 对象失败!");
18     return;
19 }
20 //设定协调层级
21 result = DD->SetCooperativeLevel(m_hWnd, DDSCL_EXCLUSIVE
22 | DDSCL_FULLSCREEN | DDSCL_ALLOWREBOOT );
23 if(result !=DD_OK)
24 {
25     MessageBox("设定程序协调层级失败!");
26     return;
27 }
28 //设定显示模式
29 result = DD->SetDisplayMode(640,480,16,0,
30                             DDSM_STANDARDVGA_MODE);
31 if(result !=DD_OK)
32 {
33     MessageBox("设定屏幕显示模式失败!");
34     return;
35 }
36 //建立主绘图页
37 memset(&DDde,0,sizeof(DDde));
38 DDde.dwSize = sizeof(DDde);
39 DDde.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
40 DDde.dwBackBufferCount = 1;
41 DDde.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE
42 | DDSCAPS_COMPLEX | DDSCAPS_FLIP;
43 result = DD->CreateSurface(&DDde,&DDSur,NULL);
44 if(result !=DD_OK)
45 {
46     MessageBox("建立主绘图页失败!");
47     return;
48 }
49 //连接后缓冲区
50 DDcaps.dwCaps = DDSCAPS_BACKBUFFER;
51 result = DDSur->GetAttachedSurface(&DDcaps,&DDBuf);
52 if(result !=DD_OK)
53 {
54     MessageBox("连接后缓冲区失败!");
55     return;
56 }
57 //声明幕后绘图区的共同特性
58 memset(&DDde,0,sizeof(DDde));
59 DDde.dwSize = sizeof(DDde);
60 DDde.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;

```

```

61 DDde.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN ;
62 }

```

### 程序说明

(1) 第 2~9 行: 声明使用 DirectDraw 所需的各个变量。第 5 行声明一个幕后暂存区数组, 数组长度依程序中所用到的位图多少而定。第 9 行声明一个颜色键结构 “key”。

(2) 第 10~56 行: 依照第 7 章里所介绍的建立 DirectDraw 步骤来运行建立 DirectDraw 对象、设定屏幕显示模式、建立主绘图页等动作。

(3) 第 58~61 行: 预先定义所有要建立的幕后绘图区的共同特性。

## 2. 建立 DirectSound

初始与建立 DirectSound 的函数为 InitDS(), 其变量声明与程序代码内容如下:

### 程序代码

```

1  //声明 DirectSound 变量
2  LPDIRECTSOUND          DS;
3  LPDIRECTSOUNDBUFFER    DSPri;
4  LPDIRECTSOUNDBUFFER    DSBuf[3];
5  WAVEFORMATEX            DSfmt;
6  WAVEFORMATEX            wfmt;
7  DSBUFFERDESC            DSde;
8  MMCKINFO                ckRiff;
9  MMCKINFO                ckInfo;
10 MMRESULT                mmresult;
11 HMMIO                   hmmio;
12 DWORD                   wsize;
13 LPVOID                   pAudio;
14 DWORD                   bytesAudio;
15 void canvasFrame::InitDS()
16
17 //建立 DirectSound 对象
18 result = DirectSoundCreate( NULL, &DS, NULL );
19 if(result != DS_OK)
20 {
21     MessageBox("建立 DirectSound 对象失败!");
22     return;
23 }
24 //设定协调层级
25 result = DS->SetCooperativeLevel( m_hwnd, DSSCL_PRIORITY );
26 if(result != DS_OK)
27 {
28     MessageBox("设定协调层级失败!");
29     return;
30 }
31 //建立主缓冲区
32 memset( &DSde, 0, sizeof(DSde) );
33 DSde.dwSize      = sizeof(DSde);

```



```

34 DSde.dwFlags      = DSBCAPS_PRIMARYBUFFER;
35 DSde.dwBufferBytes = 0;
36 DSde.lpwfxFormat   = NULL;
37 result = DS->CreateSoundBuffer( &DSde, &DSPri, NULL );
38 if(result != DS_OK)
39 {
40     MessageBox("建立主缓冲区失败!");
41     return;
42 }
43 //设定声音播放格式
44 memset( &DSfmt,0, sizeof(DSfmt) );
45 DSfmt.wFormatTag    = WAVE_FORMAT_PCM;
46 DSfmt.nChannels     = 2;
47 DSfmt.nSamplesPerSec = 44100;
48 DSfmt.wBitsPerSample = 16;
49 DSfmt.nBlockAlign   = DSfmt.wBitsPerSample / 8 * DSfmt.nChannels;
50 DSfmt.nAvgBytesPerSec = DSfmt.nSamplesPerSec * DSfmt.nBlockAlign;
51 result = DSPri->SetFormat(&DSfmt);
52 if(result != DS_OK)
53 {
54     MessageBox("设定播放格式失败!");
55     return;
56 }
57 )

```

## 程序说明

(1) 第 2~14 行: 声明使用 DirectSound 所需用到的各个变量。其中第 4 行声明一个次缓冲区声音数组, 数组长度按程序中所要播放的声音文件的多少而定。

(2) 第 15~57 行: 按照第八章所介绍的建立 DirectSound 步骤来运行建立 DirectSound 对象、建立主缓冲区、设定播放格式等动作。

## 3. 建立 DirectInput

初始化建立 DirectInput 的函数为 InitDI(), 变量声明与程序代码内容如下:

## 程序代码

```

1  //声明 DirectInput 变量
2  LPDIRECTINPUT7          DI;
3  LPDIRECTINPUTDEVICE7    DIms;
4  DIMOUSESTATE2           DImsstate;
5  void canvasFrame::InitDI()
6  {
7  //建立 DirectInput 对象
8  HINSTANCE hinst = AfxGetInstanceHandle();
9  result = DirectInputCreateEx(hinst,
10 DIRECTINPUT_VERSION, IID_IDirectInput7, (void**)&DI, NULL);
11 if(result != DI_OK)
12 {
13     MessageBox("建立 DirectInput 对象失败!");

```

```

14 return;
15 }
16 //建立输入装置对象
17 result = DI->CreateDeviceEx(GUID_SysMouse,
18 IID_IDirectInputDevice7, (void*)&DIm, NULL);
19 if(result != DI_OK)
20 {
21     MessageBox("建立鼠标输入装置失败!");
22     return;
23 }
24 //设定资料格式
25 result = DIm->SetDataFormat(&c_dfDIMouse2);
26 if(result != DI_OK)
27 {
28     MessageBox("设定资料格式失败!");
29     return;
30 }
31 //设定协调层级
32 result = DIm->SetCooperativeLevel(m_hWnd,
33 DISCL_BACKGROUND | DISCL_NONEXCLUSIVE);
34 if(result != DI_OK)
35 {
36     MessageBox("设定程序协调层级失败!");
37     return;
38 }
39 //调用输入装置
40 result = DIm->Acquire();
41 if(result != DI_OK)
42 {
43     MessageBox("调用输入装置失败!");
44     return;
45 }
46 }

```

#### 程序说明

(1) 第 2~4 行：声明使用 DirectInput 所需用到的各个变量。第 4 行声明一个代表鼠标状态的结构“DImousestate”，在后面的范例中统一把鼠标当做输入装置。

(2) 第 5~46 行：按照第九章介绍的建立 DirectSound 步骤来运行建立 DirectInput 对象、建立与调用输入装置、设定资料格式与协调层级等动作。

前面所介绍的三个建立 DirectX 的自定义函数，会在 OnCreate 函数中调用运行，以初始化与建立 DirectX 的程序架构。

### A.1.2 建立 DirectDraw 幕后暂存区

程序会为每一张要使用的位图建立一个 DirectDraw 的幕后暂存区，并将位图加载到幕后暂存区中，建立幕后暂存区的自定义函数声明为：



```
void CreateLDPla(int width,int height,char* filename,int num);
```

函数中输入的各个参数及其意义说明见表 A-1。

表 A-1

参 数	说 明
width	要加载位图的宽度，即所要建立幕后暂存区的宽度
height	要加载位图的高度，即所要建立幕后暂存区的高度
filename	要加载的位图文件名
num	幕后暂存区的索引值编号

由于在主程序的全局声明中声明一个幕后暂存区的数组 `DDPla`，因此调用此函数建立幕后暂存区时必须输入一个索引编号“`num`”，以将某一张位图加载到指定的索引值编号的幕后暂存区中，函数的内容说明如下所示。

程序代码

```

1 void canvasFrame::CreateDDPla(int width,int height,
2                               char* filename,int num)
3 {
4     //设定幕后暂存区大小
5     DDde.dwWidth = width;
6     DDde.dwHeight = height;
7     //建立幕后暂存区
8     result = DD->CreateSurface(&DDde, &DDPla[num], NULL);
9     if(result !=DD_OK)
10 {
11         MessageBox("建立幕后暂存区失败!");
12     return;
13 }
14 //加载位图
15 bitmap = (HBITMAP)::LoadImage(NULL,filename,IMAGE_BITMAP,
16                                width,height,LR_LOADFROMFILE);
17 if(bitmap==NULL)
18 {
19     MessageBox("无法加载位图!");
20     return;
21 }
22 //复制位图到绘图中
23 ::SelectObject(hdc,bitmap);
24 DDPla[num]->GetDC( &dhdh );
25 ::BitBlt( dhdh , 0 , 0 ,width,height, hdc , 0 , 0 , SRCCOPY );
26 DDPla[num]->ReleaseDC(dhdh);
27 }

```

#### 程序说明

- (1) 第 5、6 行: 按照输入的参数 width 与 height 设定幕后暂存区的大小。
- (2) 第 8~13 行: 按照 num 值建立一个幕后暂存区。
- (3) 第 15、16 行: 加载文件名为 filename 的位图到“bitmap”中。

- (4) 第 23 行: 选择位图到 hdc 中。
- (5) 第 24 行: 取得代表幕后暂存区的 DC “dhdc”。
- (6) 第 25 行: 复制 hdc 的内容到 dhdc 中, 而幕后暂存区中便存放了要使用的位图文件。
- 在调用此函数之后, 便建立了一个包含位图的幕后暂存区 DDPla[num]供程序使用。

### A.1.3 建立 DirectSound 次缓冲区

与 CreateDDPla 函数建立幕后暂存区的功能类似, CreateDSBuf 函数是用来建立 DirectSound 的次缓冲区并加载所要播放的声音文件, 函数声明如下:

```
void CreateDSBuf(char* filename,int num);
```

其中输入的各个参数及其意义说明见表 A-2。

表 A-2

参 数	说 明
filename	要加载的声音文件名
num	次缓冲区的索引值编号

同样的, 在程序全局声明中声明了一个次缓冲区数组 DSBuf, 因此调用 CreateDSBuf 函数来建立声音的次缓冲区时必须输入一个索引编号 “num”, 将要播放的声音文件加载到指定的索引编号的次缓冲区中。函数的内容说明如下:

#### 程序代码

```
1 void canvasFrame::CreateDSBuf(char* filename,int num)
2 {
3     //开启与检查文件格式
4     hnmio = mmioOpen(filename, NULL, MMIO_ALLOCBUF
5                     |MMIO_READ );
6     if(hnmio == NULL)
7     {
8         MessageBox("文件不存在!");
9         return;
10    }
11    ckRiff.fccType = mmioFOURCC('W', 'A', 'V', 'E');
12    nmresult = mmioDescend(hnmio,&ckRiff,NULL,MMIO_FINDRIFF);
13    if(nmresult != MMSYSERR_NOERROR)
14    {
15        MessageBox("文件格式错误!");
16        return;
17    }
18    ckInfo.ckid = mmioFOURCC('f', 'm', 't', ' ');
19    nmresult = mmioDescend(hnmio,&ckInfo,&ckRiff,MMIO_FINDCHUNK);
20    if(nmresult != MMSYSERR_NOERROR)
21    {
22        MessageBox("文件格式错误!");
23        return;
```

```

24 )
25 //读取文件格式
26 mmresult = mmioRead(hmmio, (HPSTR)&wfmt, sizeof(wfmt));
27 if(mmresult == -1)
28 {
29     MessageBox("读取格式失败!");
30     return;
31 }
32 mmresult = mmioAscend(hmmio, &ckInfo, 0);
33 ckInfo.ckid = mmioFOURCC('d', 'a', 't', 'a');
34 mmresult = mmioDescend(hmmio, &ckInfo, &ckRiff, MMIO_FINDCHUNK);
35 if(mmresult != MMSYSERR_NOERROR)
36 {
37     MessageBox("文件格式错误!");
38     return;
39 }
40 wsize = ckInfo.cksize;
41 //设定次缓冲区特性
42 memset(&DSde, 0, sizeof(DSde));
43 DSde.dwSize = sizeof(DSde);
44 DSde.dwFlags = DSBCAPS_STATIC | DSBCAPS_CTRLPAN
45 | DSBCAPS_CTRLVOLUME | DSBCAPS_GLOBALFOCUS;
46 DSde.dwBufferBytes = wsize;
47 DSde.lpwfxFormat = &wfmt;
48 //建立次缓冲区
49 result = DS->CreateSoundBuffer(&DSde, &DSBuf[num], NULL);
50 if(result != DS_OK)
51 {
52     MessageBox("建立次缓冲区失败!");
53     return;
54 }
55 //锁定次缓冲区并加载声音文件
56 result = DSBuf[num]->Lock(0, wsize, &pAudio, &bytesAudio,
57                             NULL, NULL, NULL);
58 if(result != DS_OK)
59 {
60     MessageBox("锁定缓冲区失败!");
61     return;
62 }
63 mmresult = mmioRead(hmmio, (HPSTR)pAudio, bytesAudio);
64 if(mmresult == -1)
65 {
66     MessageBox("读取声音文件资料失败!");
67     return;
68 }
69 result = DSBuf[num]->Unlock(pAudio, bytesAudio, NULL, NULL);
70 if(result != DS_OK)
71 {
72     MessageBox("解除锁定缓冲区失败!");

```

```

73 return;
74 }
75 mmioClose(hmmio, 0);
76 }

```

#### 程序说明

(1) 第 4~40 行：主要是按输入的声音文件名 filename 打开文件，并进行检查与读取文件格式的动作。

(2) 第 42~47 行：设定要建立的次缓冲区特性。

(3) 第 49~54 行：按照 num 值建立一个次缓冲区。

(4) 第 56~74 行：锁定次缓冲区，并加载该次缓冲区中所要播放的声音文件。

(5) 第 75 行：关闭所有打开的声音文件。

在调用此函数之后，便建立了一个包含声音文件的次缓冲区 DSBuf[num]，可用在程序中播放声音。

### A.1.4 设定颜色键函数

最后再来看看设定颜色键的自定义函数 ColorKey，此函数的声明为：

```
void ColorKey(int num);
```

函数中输入的参数 num 是设定颜色键的幕后暂存区索引值编号，如设定颜色键的幕后暂存区为 DDPla[1]，调用此函数时便输入“1”。内容如下所示。

#### 程序代码

```

1 void canvasFrame::ColorKey(int num)
2 {
3     key.dwColorSpaceHighValue = 0;
4     key.dwColorSpaceLowValue = 0;
5     DDPla[num]->SetColorKey(DDCKEY_SRCBLT, &key);
6 }

```

#### 程序说明

(1) 第 3、4 行：在后面的范例里，将黑色当作颜色键，所以会发现若是空的位图，四周的颜色都是黑色，程序代码是设定黑色为颜色键（高位与低位皆为 0）。

(2) 第 5 行：根据输入 num 的值设定幕后暂存区的颜色键。

在看完了上述的几个范例所会用到的自定义函数的说明之后，接下来开始实际制作几个有趣的游戏吧。

## A.2 绚丽的电流急急棒

这一节里的电流急急棒范例使用 DirectX 制作全屏的画面，并且加入极具震撼力的音效。

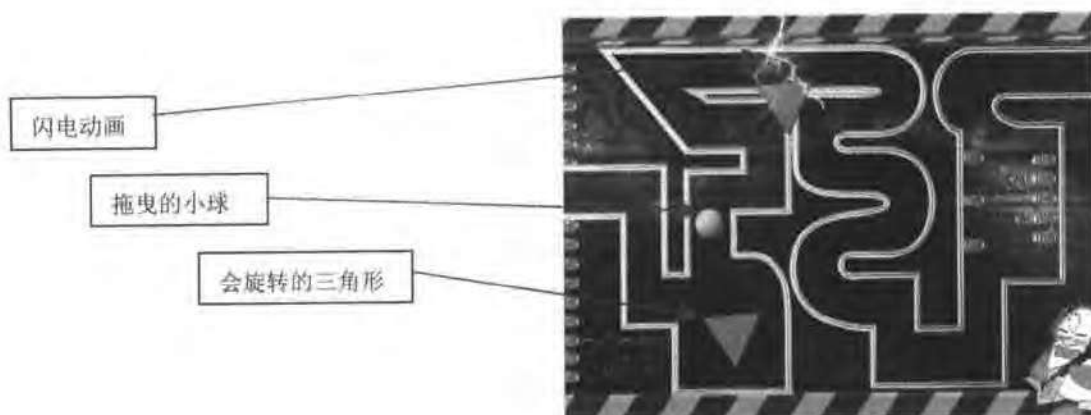
### A.2.1 游戏功能介绍

下面来看看这个范例程序运行时画面，如图 A-1~图 A-4 所示。



程序执行初始化并加载所需的文件

图 A-1



闪电动画

拖曳的小球

会旋转的三角形

图 A-2



小心不要碰到旁边的圆柱

图 A-3

到达这里便过关了。按下 F1 键  
可以重新开始。



图 A-4

若小球碰到了转动的三角形或者边缘四周便会出现失败信息，如图 A-5 所示。



图 A-5

在看过了程序的功能之后，下一小节将解说这个游戏画面中动画产生以及碰撞检测的方式。

## A.2.2 游戏功能设计方法

设计这个电流急急棒游戏的重点主要有两个，分别为：

- (1) 画面上动画的产生。
- (2) 检测小球是否碰撞到四周边缘或者是转动的三角形。

下面说明这两种功能的制作方式。

### 1. 动画的产生

在这个游戏中，使用了连续的 8 张前景图来呈现玩家所看到的画面，利用不间断地轮流显示这 8 张图的方式来产生动画效果，动画效果包含三角形的转动以及闪电动画。



## 2. 检测碰撞

每一张前景图都有其对应的背景图，下面便是一张前景图及与其对应的背景图，如图 A-6 和图 A-7 所示。



图 A-6 前景图



图 A-7 背景图

当小球移动的时候，程序会判断小球四周的“碰撞点”是否进入背景图中的黑色区域。若任一碰撞点所在位置点的颜色为黑色，即表示发生碰撞。

小球周围的碰撞点是自己设定的，利用一些绘图软件（如 PhotoImpact）便可找出小球圆周上点的位置，如图 A-8 所示。

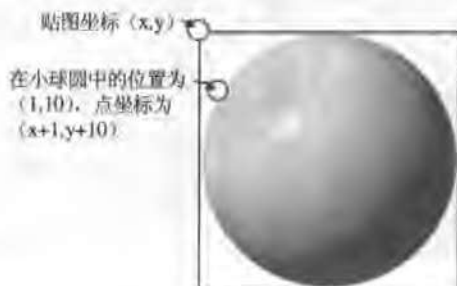


图 A-8

如图 A-8 中，若小球左上角的贴图点坐标为  $(x, y)$ ，而碰撞点在位图中的位置为  $(m, n)$ ，则碰撞点的坐标为  $(x+m, y+n)$ 。在这个范例中，给定了 16 个碰撞点，利用这些碰撞点可以非常精准地判断出是否发生碰撞。

### A.2.3 程序内容说明

在这个程序中先要建立一个定时器，除玩家按下【Esc】键结束游戏外，这个定时器会不断地发出 WM\_TIMER 信息来重绘屏幕并取得鼠标的输入信息。

程序中利用三个布尔变量“collide”、“win”、“over”来判断玩家控制的小球是否“发生碰撞”、“过关”或是“游戏结束”。当玩家的小球发生碰撞或者过关时，“over”变量为 true，此时程序并不会读取鼠标的输入信息，只是不断地重绘屏幕，而屏幕上的小球也会一直出现在同样的地方；当“over”变量为 false 时，程序便读取玩家通过鼠标所输入的信息，并重设小球的贴图坐标，控制小

球的移动。接下来就这个范例程序的部分代码进行说明。

在建构中需要加入一些主要的动作,如显示【加载中】图标与建立各个存储位图的幕后暂存区,以及播放声音的次缓冲区,并且将检测碰撞用的背景图片所有点的颜色值读入各个数组中。下面来看看程序代码的内容说明。

#### 程序代码

```

1  HDC hdc,dhdc;
2  HBITMAP bitmap;
3  int a,i,j,t,num;
4  int x=1,y=190;
5  int bg[8][640][480];
6  int px[16];
7  int py[16];
8  BOOL collide,over,win;
9  char str[10];
10 canvasFrame::canvasFrame()
11 {
12 Create(NULL,"绘图窗口",WS_POPUP);
13 hdc = ::CreateCompatibleDC(NULL);
14 ::ShowCursor(false);
15 CreatedDPla(640,480,"load.bmp",19);
16 DDBuf->BltFast( 0 , 0 , DDPla[19], CRect(0,0,640,480) ,
17             DDBLTFAST_WAIT);
18 DDSur->Flip( NULL , DDFLIP_WAIT );
19 for(i=0;i<=7;i++) //加载前景图与背景图
20 {
21 sprintf(str,"bgbmp%d.bmp",i);
22 CreatedDPla(640,480,str,i);
23 sprintf(str,"mbgbmp%d.bmp",i);
24 CreatedDPla(640,480,str,i+8);
25 }
26 CreatedDPla(34,34,"ball.bmp",16); //加载小球图
27 CreatedDPla(638,180,"win.bmp",17); //加载过关图
28 CreatedDPla(638,180,"lose.bmp",18); //加载失败图
29 ColorKey(16);
30 ColorKey(17);
31 ColorKey(18);
32 for(a=0;a<=7;a++)
33 {
34 DDPla[a+8]->GetDC( &hdc );
35 for(i=0;i<640;i++)
36     for(j=0;j<480;j++)
37         bg[a][i][j] = ::GetPixel(hdc,i,j);
38 DDPla[a+8]->ReleaseDC(hdc);
39 }
40 CreateDSBuf("s0.wav",0); //加载背景音乐
41 CreateDSBuf("s1.wav",1); //加载碰撞音乐
42 CreateDSBuf("s2.wav",2); //加载过关音乐

```

```
43 DSBuf[0]->Play(0,0,1);
44 }
```

## 程序说明

(1) 第 1~9 行：全局变量声明，其中第 4 行的程序代码所声明的变量  $x$  与  $y$  为小球贴图的坐标点，初始值位置设为 (1,190)；第 5 行的程序代码声明所有存储背景图颜色的三阶数组；第 6、7 行的程序代码则声明代表小球碰撞点坐标的数组  $px$  与  $py$ 。

(2) 第 14 行：取消鼠标的显示。

(3) 第 15 行：调用 CreateDDPla 函数来建立一个幕后暂存区并加载【加载中】图标。

(4) 第 16~18 行：在屏幕上显示此图标画面。

(5) 第 19~25 行：建立各个幕后暂存区并加载前景图 (DDPla[0]~DDPla[7]) 与背景图 (DDPla[8]~DDPla[15])。

(6) 第 26~31 行：加载其他所需的位图并调用 ColorKey 函数来设定颜色键。

(7) 第 32~39 行：取得每一张背景图的所有像素点颜色并存入  $bg$  数组中，例如第一张背景图坐标为 (10,100)，像素点的颜色值即存储在  $bg[0][10][100]$  中。利用这样的存储像素点颜色的方法，在后面可以直接输入碰撞点所在的坐标，并判断该点所在的位置是否为黑色 (即是否发生碰撞)。

(8) 第 40~43 行：调用 CreateDSBuf 函数加载程序中要播放的声音文件。第 43 行程序代码播放 DSBuf[0] 来产生背景音乐。

接下来在 OnTimer 函数中判断代表游戏结束的 over 变量值是 true 或 false，如果为 false，则运行 Playing 函数，否则运行 Over 函数。OnTimer 函数的内容说明如下。

## 程序代码

```
1 void canvasFrame::OnTimer(UINT nIDEvent)
2 {
3     if(!over)
4         Playing();           //游戏正在进行中
5     else
6         Over();              //过关或发生碰撞
7     DDSur->Flip( NULL , DDFLIP_WAIT );
8     CFrameWnd::OnTimer(nIDEvent);
9 }
```

## 程序说明

第 7 行：运行 Playing 或 Over 函数后完成缓冲区的贴图，将最后结果翻页显示于屏幕上。

Playing 函数中主要功能是依次显示各张前景图产生动画效果，并不断检测玩家由鼠标所输入的信息，并据此重设小球贴图的坐标，然后利用碰撞点进行碰撞检测，以下是其程序说明。

## 程序代码

```
1 void canvasFrame::Playing()
2 {
3     t++;
4     if(t%5 == 0)
5     {
6         num++;
7         if(num == 8)
8             num = 0;
```

```
9   t=0;
10  }
11  DDBuf->BltFast( 0 , 0 , DDPla[num], CRect(0,0,640,480) ,
12               DDBLTFAST_WAIT);
13  result = DIm->GetDeviceState(sizeof(DImstate),
14                               (LPVOID)&DImstate);
15  if(result != DI_OK )
16  {
17   MessageBox("取得鼠标状态失败!");
18   return;
19  }
20  x += DImstate.lX;
21  y += DImstate.lY;
22  if(x<=1)
23   x=1;
24  if(x>=605)
25   x=605;
26  if(y<=20)
27   y=20;
28  if(y>=430)
29   y=430;
30  px[0] = x+16;
31  px[1] = x;
32  px[2] = x+33;
33  px[3] = x+16;
34  px[4] = x+5;
35  px[5] = x+28;
36  px[6] = x+27;
37  px[7] = x+5;
38  px[8] = x+2;
39  px[9] = x+9;
40  px[10] = x+31;
41  px[11] = x+23;
42  px[12] = x+24;
43  px[13] = x+30;
44  px[14] = x+9;
45  px[15] = x+1;
46  py[0] = y+0;
47  py[1] = y+16;
48  py[2] = y+16;
49  py[3] = y+33;
50  py[4] = y+5;
51  py[5] = y+5;
52  py[6] = y+29;
53  py[7] = y+28;
54  py[8] = y+24;
55  py[9] = y+31;
56  py[10] = y+23;
57  py[11] = y+31;
```

```

58 py[12] = y+2;
59 py[13] = y+8;
60 py[14] = y+2;
61 py[15] = y+10;
62 DDBuf->BltFast(x,y,DDPla[16],CRect(0,0,34,34),
63     DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);
64 for(i=0;i<=15;i++)
65     if(bg[num][px[i]][py[i]]==0)                //发生碰撞
66     {
67         DDBuf->BltFast(0,150,DDPla[18],CRect(0,0,638,180),
68             DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);
69         over = true;
70         DSBuf[1]->Play(0,0,0);
71         collide = true;
72         break;
73     }
74 if(px[2] > 630)                                //过关
75 {
76     DDBuf->BltFast(0,150,DDPla[17],CRect(0,0,638,180),
77         DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);
78     DSBuf[0]->Stop();
79     win = true;
80     over = true;
81     DSBuf[2]->Play(0,0,1);
82 }
83 }

```

### 程序说明

(1) 第3~10行: 每次累加变量  $t$  的值, 当  $t\%5==0$  时, 将图号  $num$  加1。在此条件式中, 可以改变除数的大小来控制动画显示的速度。

(2) 第11行: 根据  $num$  的值在后缓冲区中贴入要显示的前景图。

(3) 第13~19行: 取得鼠标的输入状态。

(4) 第20、21行: 依照鼠标移动的量重新计算小球的贴图坐标  $x$  与  $y$ 。

(5) 第22~29行: 设定小球贴图的坐标点只能在左上角点(1,20)与右下角点(605,430)的区域中。

(6) 第30~61行: 依照小球贴图的坐标点( $x,y$ )来设定其各个碰撞点的坐标, 例如, 第一个碰撞点是( $px[0],py[0]$ ), 其坐标为( $x+16,y+0$ )。

(7) 第62行: 根据贴图的坐标( $x,y$ )来贴上小球图。

(8) 第64~73行: 在  $for$  循环中以小球的16个碰撞点判断是否发生碰撞。

(9) 第65行: 判断碰撞点所在位置上背景图的颜色是否为黑色, 若为黑色, 就表示发生碰撞, 判断式为“ $if(bg[num][px[i]][py[i]]==0)$ ”。前面已经将每一张背景图所有像素点的颜色存入数组  $bg$  中。若目前显示的为第一张前景图, 且碰撞点的坐标为(1,10), 那么在第一张背景图中, 该坐标点的颜色便是  $bg[0][1][10]$ , 如果为0则代表黑色, 即小球与三角形或是四周的边缘发生碰撞。

(10) 第67~72行: 在小球发生碰撞后, 其程序代码会贴上失败的信息, 并将  $collide$  与  $over$  变量设为  $true$ , 播放发生碰撞的声音, 然后结束循环。

(11) 第74行: 如果小球未发生碰撞, 则其程序代码以第三个碰撞点的  $X$  坐标  $px[2]$  来判断是

否过关。若小球已到达终点便如图 A-9 所示。

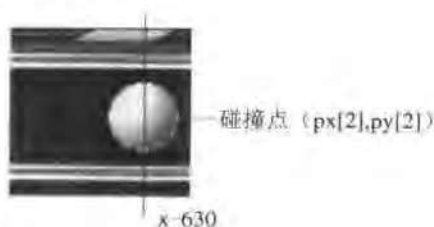


图 A-9

这里设定  $px[2]$  大于 630 过关。

(12) 第 76~81 行: 当小球到达终点后, 接下来的程序代码会贴上过关的信息, 将 `win` 与 `over` 变量设为 `true`, 然后播放过关音乐。

#### 程序代码

```
1 void canvasFrame::Over()
2 {
3     t++;
4     if(t%10 == 0)
5     {
6         num++;
7         if(num == 8)
8             num = 0;
9         t=0;
10    }
11    DDBuf->BltFast( 0 , 0 , DDPla[num], CRect(0,0,640,480) ,
12                  DBLTFAST_WAIT);
13    DDBuf->BltFast(x,y,DDPla[15],CRect(0,0,34,34),
14                  DBLTFAST_WAIT|DBLTFAST_SRCCOLORKEY);
15    if(win)
16        DDBuf->BltFast(0,150,DDPla[17],CRect(0,0,638,180),
17                      DBLTFAST_WAIT|DBLTFAST_SRCCOLORKEY);
18    if(collide)
19        DDBuf->BltFast(0,150,DDPla[18],CRect(0,0,638,180),
20                      DBLTFAST_WAIT|DBLTFAST_SRCCOLORKEY);
21 }
```

#### 程序说明

(1) 第 13、14 行: 贴上小球图, 此时贴图坐标  $x$  与  $y$  已不会改变, 因此程序会一直把小球贴到固定的位置上。

(2) 第 14~18 行: 判断式根据变量 `win` 与 `collide` 判断游戏结束的状况, 若为过关则贴上过关信息, 若为小球发生碰撞则贴上失败信息。

在这个程序中, 设计两个可以按下运行的功能, 一是按下【Esc】键结束程序, 另一个是按下【F1】键重新开始游戏。处理玩家按下按键的函数为 `OnKeyDown`, 下面来看其程序说明。

#### 程序代码

```
1 void canvasFrame::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
```

```

2
3  if( nChar== VK_ESCAPE )           //按下【Esc】键
4      PostMessage(WM_CLOSE );       //结束程序
5  if( nChar == VK_F1)               //按下【F1】键
6  {
7      over = false;
8      if(win)
9      {
10         win = false;
11         DSBuf[2]->Stop();
12         DSBuf[0]->Play(0,0,1);
13     }
14     else
15         collide = false;
16     x = 1;
17     y = 190;
18 }
19 CFrameWnd::OnKeyDown(nChar, nRepCnt, nFlags);
20

```

## 程序说明

(1) 第 3~4 行：判断玩家是否按下了【Esc】键，若是，则传送 WM\_CLOSE 信息结束游戏。

(2) 第 5~15 行：判断玩家是否按下了【F1】键来重新开始游戏，若是，则将 over 设为 false。如果目前的游戏状况为玩家过关（第 9 行程序代码），则将 win 设为 false，并停止播放过关音乐，改播背景音乐；而如果是玩家失败，则将 collide 设为 false。

(3) 第 16、17 行：将小球的坐标移回起点，便可重新进行游戏。

上述便是这个电流急急棒游戏的制作方式以及程序内容说明。各位不妨试着来玩玩这个游戏，测试一下自己控制鼠标的稳定度跟注意力。

## A.3 太空射击游戏

您有没有玩过雷电、沙罗曼蛇等知名射击游戏？如果您玩过相信您一定忘不了那身临其境且具有震撼力的射击快感。在这一节里我们将亲自动手制作一个射击游戏，感受一下歼灭太空怪物的乐趣！

### A.3.1 游戏功能介绍

在这个射击游戏中，共设计了三种类型的太空怪物，它们都有各自的特色，游戏的目的是通过不断地射击这些怪物来获得分数。随着分数越来越高，怪物产生的速度便会越来越快，从而增加游戏的难度，下面先来看看这个范例运行时的画面，如图 A-10~图 A-13 所示。



程序执行初始化动作并加载相关的图文件

图 A-10

利用鼠标来移动飞机，或按下鼠标左键来发射飞弹

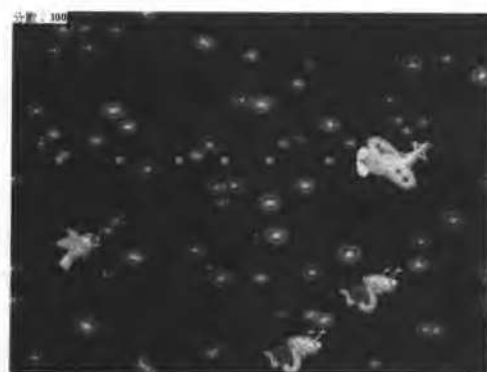
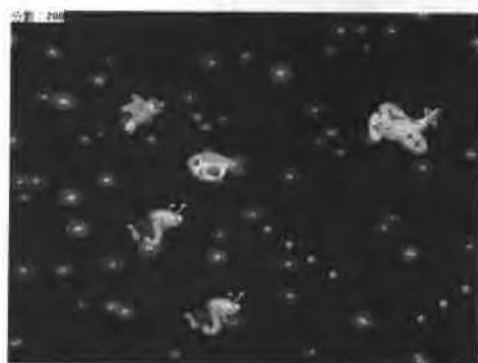


图 A-11



怪物出现，并且攻击飞机

图 A-12

飞机被怪物击中，飞机就会爆炸



图 A-13



看过了这个射击游戏的功能与运行画面之后，下一小节里就来介绍这个范例中使用的一些设计的概念与技巧。

## A.3.2 滚动背景的设计

在这一范例中使用了较多的技巧，包括滚动的背景、不同怪物的产生与它们各自不同的行为、检测碰撞的方法以及飞机与怪物会发射子弹等，接下来分别讨论这些功能的设计方法。

“滚动背景”的设计方法之前就已经讨论过了。同样在这个程序中，使用了一张  $640 \times 480$  的背景图，而每次进行背景的贴图时分为两次，请看图 A-14 的说明。

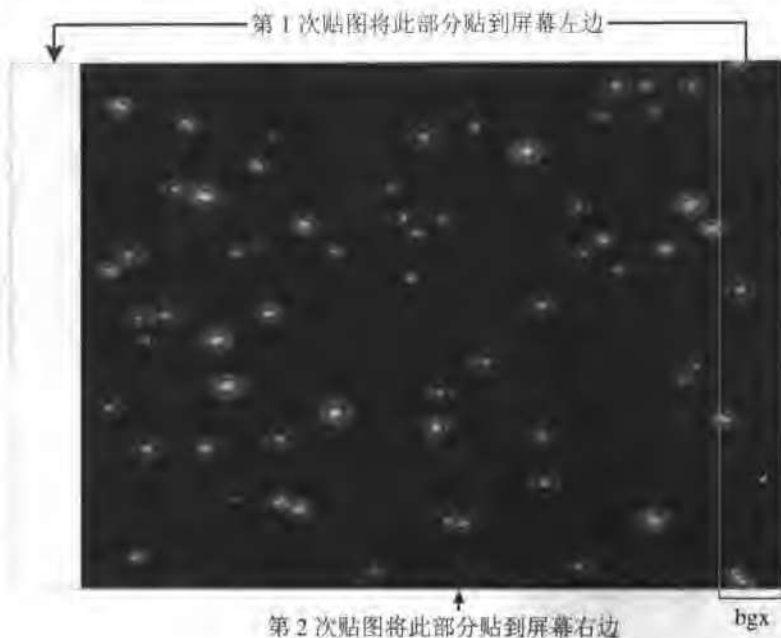


图 A-14

在图 A-15 所示中，背景图外围的方框代表显示屏幕的窗口，“bgx”为右边部分要贴到左边窗口的图形宽度，每次定时器在运行的时候，bgx 的大小就会递增 10，以便于产生背景由左向右滚动的效果。当 bgx 等于屏幕的宽度 640 时，将 bgx 重设为 0。

## A.3.3 怪物的产生与移动

在这个范例中，程序会不断地依随机数产生三种不同的怪物，下面就来说明这三种怪物的产生与移动方式。

### 1. 发射子弹的蜗牛

三种怪物的产生是依照条件式 “rand()%3” 来判断，当结果为 0 时便产生会发射子弹的蜗牛，其行进方向在 X 轴方向为每次向前 20 个像素，Y 轴方向则按照 “rand()%2” 的结果来决定往上或往下 20 像素，并且限制其在 Y 轴方向上的移动位置介于窗口的 0~480 之间。当蜗牛在 X 轴上的贴图

坐标小于 100 时，每次移动都会发射一颗子弹。

## 2. 追踪目标的小鸟

小鸟会根据飞机的所在位置而自动朝飞机前进。让怪物追逐目标的方法我们也在前面提过，其算法大致如下所示。

```
if(小鸟.x > 飞机.x)
    小鸟.x-=30;
else
    小鸟.x+=30;
if(小鸟.y > 飞机.y)
    小鸟.y-=30;
else
    小鸟.y+=30;
```

## 3. 从后面出现的鱼

从后面出现的鱼每次产生时其所在的 X 坐标为 580，而后每次向左移动 20 个像素点，即鱼的贴图坐标每次都递减 20。

了解了这几种程序中出现的怪物的特性之后，接下来看看游戏中子弹的产生方法。

### A.3.4 子弹的产生

在这个程序中，当玩家按下鼠标左键后，飞机便会发射一颗子弹，同样蜗牛也会发射子弹，每一颗子弹便是一个粒子。以飞机的子弹为例，其结构如下：

```
struct bullet          //飞机的子弹
{
    int x;              //子弹贴图的 x 坐标
    int y;              //子弹贴图的 y 坐标
    int hitx;           //子弹碰撞点的 x 坐标
    int hity;           //子弹碰撞点的 y 坐标
    BOOL exist;         //子弹是否存在
};
```

每一颗子弹都有其贴图的坐标位置 (x,y) 与代表碰撞点的坐标 (hitx,hity)，由于子弹的面积很小，因此仅以一个碰撞点来代表该子弹。当碰撞点进入了怪物的区域范围内，则代表子弹击中了怪物。

同样的道理，如果蜗牛所发射子弹的碰撞点进入了飞机的区域范围内，也就表示子弹击中了飞机。

### A.3.5 检测碰撞的方法

在这个程序中所要检测碰撞的状况有下列三种：

- (1) 飞机的子弹打到怪物；

(2) 怪物的子弹打到飞机;

(3) 飞机撞到了怪物,

上述这些状况的碰撞检测方法, 都是利用碰撞点判断是否在其他物体所在的区域中来决定是否发生碰撞, 下面分别就检测子弹是否命中目标以及飞机与怪物发生撞击来做说明。

## 1. 子弹命中目标

每一种物体都必须设定其代表的区域。以怪物来说, 将每一只怪物都划分为三个区块来代表该怪物的区域范围, 如图 A-15 所示。

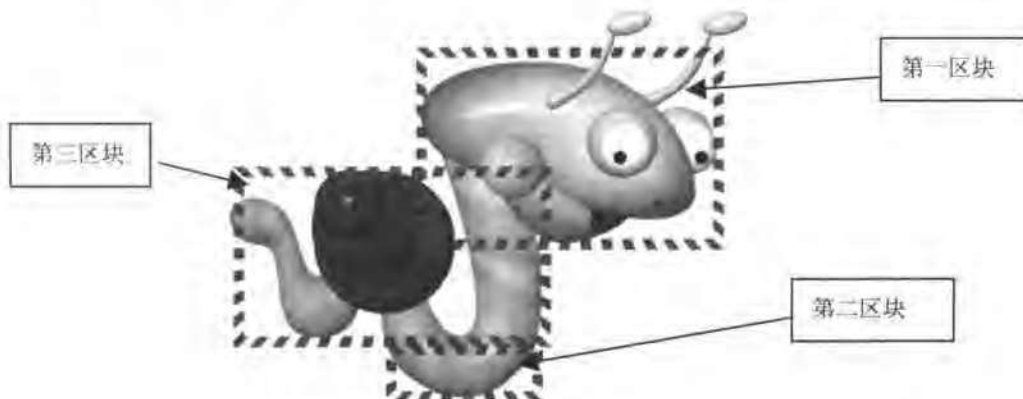


图 A-15

同样可以在绘图软件中找出每一个区块的左上点与右下点的坐标, 如此便可以确认代表怪物的每一个区块范围。当有任何碰撞点进入某一区块中, 即表示发生碰撞, 如图 A-16 所示。

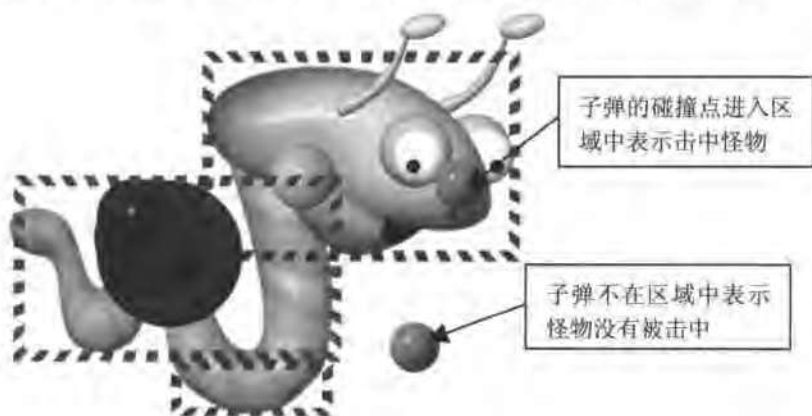


图 A-16

由图 A-16 可以看出, 利用三个区块来代表怪物所在的区域, 已经可以很精确的判断子弹是否打中怪物。当然也可以划分出更小更多的区块来代表一个物体的区域范围, 以便增加碰撞检测的准确度。

## 2. 飞机撞击怪物

除了定义每一只怪物的区域范围来判断子弹是否打中怪物之外, 还必须为怪物设定碰撞点用来

检测是否与飞机发生撞击，如图 A-17 所示。

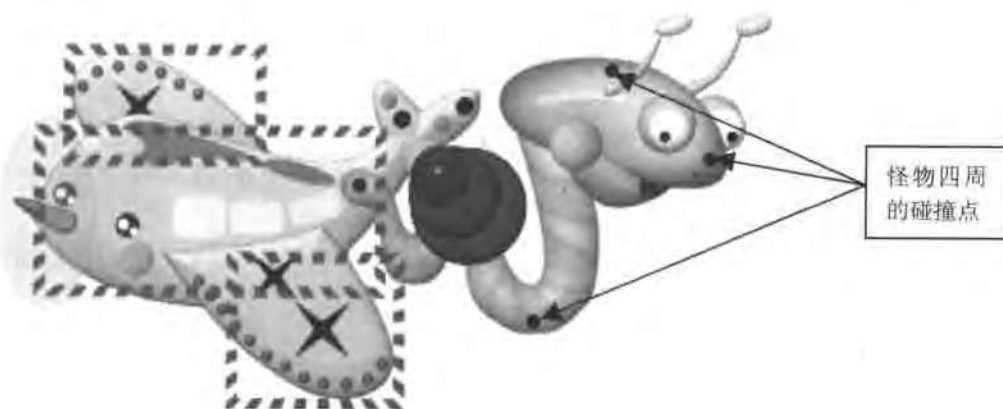


图 A-17

在这个范例中，我们为怪物设定了四个碰撞点，当任何一个碰撞点进入飞机所代表的区域中，便表示与飞机发生碰撞，如图 A-18 和图 A-19 所示。

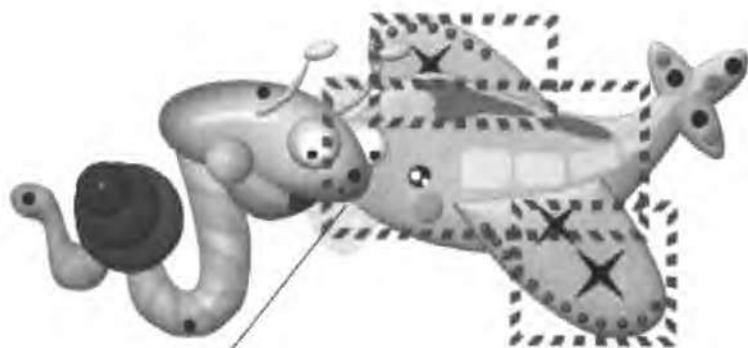


图 A-18

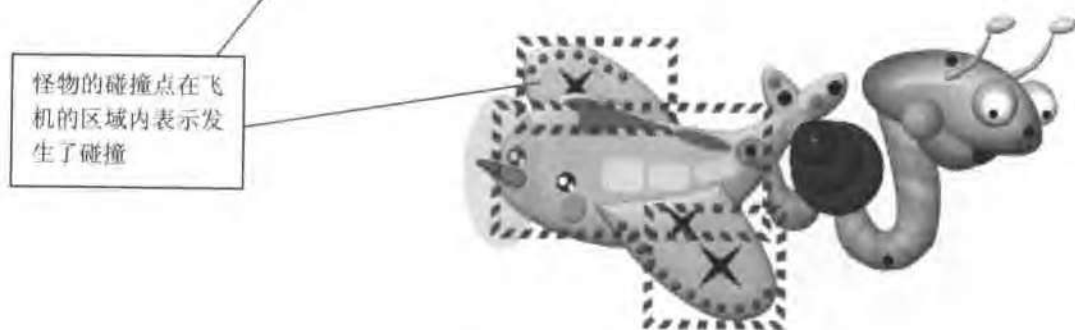


图 A-19

同样，如果将代表飞机的区域划分的越多越细，而且增加怪物四周碰撞点的数量，将可以更准确地检测碰撞。

## A.3.6 程序编写的方法

在这个程序中将以一个定时器来模拟游戏的循环。随着定时器的运行，程序会不断地读取鼠标的输入信息，并且重绘屏幕内容以及产生怪物。

使用一个布尔变量 `over` 来表示目前游戏的状态，当飞机被怪物的子弹打中或者发生撞击而结束游戏时，`over` 会设为 `true`。若玩家按下【F1】键，`over` 则设为 `false`，便可重新进行游戏，接下来看看这个范例的程序内容与说明。

现在来看一下这个程序中所定义的几个结构内容。怪物与飞机的子弹在前面已经说明，您应该可以了解其中定义成员的内容。

### 程序代码

```

1  struct bullet           //飞机的子弹
2  {
3      int x;              //贴图的 x 坐标
4      int y;              //贴图的 y 坐标
5      int hitx;           //碰撞点的 x 坐标
6      int hity;           //碰撞点的 y 坐标
7      BOOL exist;         //子弹是否存在
8  };
9  struct cartridge        //怪物的子弹
10 {
11  int x;
12  int y;
13  int hitx;
14  int hity;
15  BOOL exist;
16  };
17 struct ship             //飞机
18 {
19  int x;                  //贴图的 x 坐标
20  int y;                  //贴图的 y 坐标
21  RECT r[3];             //飞机的区块
22  };
23 struct monster          //怪物
24 {
25  int x;                  //贴图的 x 坐标
26  int y;                  //贴图的 y 坐标
27  int type;              //怪物的类型
28  int draw;              //控制贴图速度的变量
29  RECT r[3];             //怪物的区块
30  POINT p[4];            //怪物的碰撞点
31  BOOL exist;            //怪物是否存在
32  };

```

### 程序说明

(1) 第 17~22 行：飞机结构除了定义贴图坐标 `x`、`y` 外，还定义了代表区域的一个大小为 3 的 `RECT` 数组“`r`”。

(2) 第 23~32 行: 在怪物的结构中定义了代表怪物类型的 type、代表区域的 RECT 数组“r”、以及碰撞点数组“p”。

#### 程序代码

```

1  HDC hdc,dhdc;
2  HBITMAP bitmap;
3  int i,j,m,n;
4  int bgx;
5  int ccount,bcount,mcount;
6  int t,f;           //控制屏幕贴图
7  int score=0;       //分数
8  BOOL press,hit,over;
9  cartridge c[100];  //怪物子弹数组
10 bullet b[100];     //飞机子弹数组
11 monster mon[50];   //怪物数组
12 ship ship;
13 char str[20];
14 canvasFrame::canvasFrame()
15 {
16 Create(NULL,"绘图窗口",WS_POPUP);
17 hdc = ::CreateCompatibleDC(NULL);
18 ::ShowCursor(false);
19 CreatedDPLa(640,480,"load.bmp",9);
20 DDBuf->BltFast( 0 , 0 , DDPLa[9], CRect(0,0,640,480) ,
21               DDBLTFAST_WAIT);
22 DDSur->Flip( NULL , DDFLIP_WAIT );
23 /*建立各个幕后暂存区与次缓冲区的程序代码(略)*/
24 ship.x = 540;
25 ship.y = 200;
26 DSBuf[0]->Play(0,0,1);    //播放背景音乐
27 }
```

#### 程序说明

(1) 第 9~11 行: 分别声明怪物子弹数组“cartridge”、飞机子弹数组“bullet”与怪物数组“mon”。第 5 行的程序代码中声明的变量“ccount”、“bcount”、“mcount”则分别是用来记录各个数组中还存在的怪物子弹、飞机子弹与怪物的数量。

(2) 第 24、25 行: 初始化飞机开始时的贴图位置。

每次在 OnTimer 函数运行的时候, 会进行所有屏幕上的贴图、产生怪物、取得鼠标输入信息等许多动作。下面来看看其中的程序代码内容与说明。

#### 程序代码

```

1  void canvasFrame::OnTimer(UINT nIDEvent)
2  {
3  if(!over)
4  {
5  t++;
6  if(score>10000)
7  f = 20;
```

```

8  else if(score<=10000 && score>5500)
9      f = 30;
10 else if(score<=5500 && score >2000)
11     f = 40;
12 else
13     f = 50;
14 if(t%f==0)
15 {
16     Generate();                //产生怪物
17     t=0;
18 }
19 result = DImgs->GetDeviceState(sizeof(DImstate), (LPVOID)&DImstate);
20 if(result != DI_OK )
21 {
22     MessageBox("取得鼠标状态失败!");
23     return;
24 }
25 DDBuf->BlitFast( bgx , 0 , DDPla[0], CRect(0,0,640-bgx,480) ,
26                 DDBLTFAST_WAIT);
27 DDBuf->BlitFast( 0 , 0 , DDPla[0], CRect(640-bgx,0,640,480) ,
28                 DDBLTFAST_WAIT);
29 ShipMove();
30 PasteMon();
31 Shoot();
32 if(ccount !=0)                //贴上怪物子弹
33     for(i=0;i<100;i++)
34         if(c[i].exist)
35             DDBuf->BlitFast(c[i].x,c[i].y,DDPla[6],CRect(0,0,10,10),
36                             DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);
37 if(bcount != 0)                //贴上飞机子弹
38     for(i=0;i<100;i++)
39         if(b[i].exist)
40             DDBuf->BlitFast(b[i].x,b[i].y,DDPla[2],CRect(0,0,10,10),
41                             DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);
42 MonBombShip();
43 HitShip();
44 HitMon();
45 bgx+=10;
46 if(bgx==640)
47     bgx=0;
48 DDBuf->GetDC( &hdc );
49 sprintf(str,"分数: %d",score);
50 ::TextOut(hdc, 0, 0,str, lstrlen(str));    //显示分数
51 DDBuf->ReleaseDC( hdc);
52 DDSur->Flip( NULL , DDFLIP_WAIT );
53 }
54 CFrameWnd::OnTimer(nIDEvent);
55 }

```

**程序说明**

- (1) 第 3 行: 判断若目前游戏的状态并未结束, 则运行 OnTimer 中的程序代码。
- (2) 第 6~13 行: 判断式按照目前玩家所得的分数来改变产生怪物的间隔变量“f”的值。f 值越小, 产生怪物的速度越快。
- (3) 第 14 行: 判断条件式“t%f==0”是合为真。若为真, 则调用调用 Generate()函数来产生怪物, t 值为一个记录 OnTimer 函数运行次数的变量, 当产生怪物后此变量重设为 0。
- (4) 第 19~24 行: 取得鼠标的输入信息。
- (5) 第 25~28 行: 分两次贴图产生滚动的背景图。
- (6) 第 29 行: 调用 ShipMove()函数, 按照鼠标的移动重设飞机的贴图坐标, 完成飞机移动后的贴图。
- (7) 第 30 行: 调用 PasteMon()函数贴上所有还存活的怪物。
- (8) 第 31 行: 调用 Shoot()函数处理当玩家按下鼠标左键发射子弹的动作。
- (9) 第 32~36 行: 判断怪物子弹数组 c 中还存在的子弹, 依其坐标(c[i].x, c[i].y)将子弹贴于后缓冲区中。
- (10) 第 37~41 行: 按照相同的方式贴上飞机所发射的子弹。
- (11) 第 42 行: 调用 MonBombShip()函数判断怪物是否与飞机发生碰撞。
- (12) 第 43 行: 调用 HitShip()函数判断怪物的子弹是否打中飞机。
- (13) 第 44 行: 调用 HitMon()函数判断飞机的子弹是否打中怪物。
- (14) 第 48~50 行: 在绘图页上显示分数字符串。

在 OnTimer 函数中, 调用了许多个自定义函数来运行各个程序的功能, 下面来看看产生怪物的 Generate()函数中的内容。

**程序代码**

```

1 void canvasFrame::Generate()
2 {
3     switch(rand()%3)           //以随机数产生怪物
4     {
5         case 0:                 //产生蜗牛
6             for(i=0;i<50;i++)
7             {
8                 if(!mon[i].exist)
9                 {
10                     mon[i].type = 0;
11                     mon[i].x = 0;
12                     mon[i].y = rand()%420;
13                     mon[i].exist = true;
14                     mon[i].r[0].left = mon[i].x + 28;
15                     mon[i].r[0].top = mon[i].y + 7;
16                     mon[i].r[0].right = mon[i].x + 72;
17                     mon[i].r[0].bottom = mon[i].y + 34;
18                     mon[i].r[1].left = mon[i].x;
19                     mon[i].r[1].top = mon[i].y + 23;
20                     mon[i].r[1].right = mon[i].x + 49;
21                     mon[i].r[1].bottom = mon[i].y + 59;
22                     mon[i].r[2].left = mon[i].x + 24;

```



//产生小鸟

//产生角

**程序说明**

- (1) 第 3 行: 按照条件式 “rand()%3” 的结果来决定产生何种怪物。
- (2) 第 8 行: for 循环会在数组 mon 中寻找 mon[i].exist 不为 true 的元素, 即该元素中的怪物不存在, 然后第 10~33 行的程序代码便设定该元素中怪物的各项特性。
- (3) 第 10~12 行: 设定 “mon[i].type=0” 代表该 mon 数组元素中的怪物是蜗牛, 其初始的 X 坐标为 0, Y 坐标为 rand()%420, 该怪物便会从最左边的任意高度上出现。
- (4) 第 13 行: 将 “mon[i].exist” 设为 true。
- (5) 第 14~25 行: 按照怪物的种类来设定其各个区块的左上角和右下角的坐标。
- (6) 第 26~33 行: 设定怪物碰撞点的坐标。
- (7) 第 34 行: 累加目前存活怪物的数量。

**程序代码**

```

1  void canvasFrame::ShipMove()
2  {
3  ship.x += DIMstate.lX;
4  ship.y += DIMstate.lY;
5  if(ship.x<0)                //是否已至左边界
6  ship.x = 0;
7  if(ship.x>540)              //是否已至右边界
8  ship.x = 540;
9  if(ship.y<0)                //是否已至上边界
10 ship.y = 0;
11 if(ship.y>406)              //是否已至下边界
12 ship.y = 406;
13 ship.r[0].left = ship.x + 16;
14 ship.r[0].top = ship.y + 3;
15 ship.r[0].right = ship.x + 50;
16 ship.r[0].bottom = ship.y + 53;
17 ship.r[1].left = ship.x + 8;
18 ship.r[1].top = ship.y + 16;
19 ship.r[1].right = ship.x + 78;
20 ship.r[1].bottom = ship.y + 50;
21 ship.r[2].left = ship.x + 35;
22 ship.r[2].top = ship.y + 35;
23 ship.r[2].right = ship.x + 81;  *
24 ship.r[2].bottom = ship.y + 71;
25 DDBuf->BlitFast(ship.x,ship.y,DDPla[1],CRect(0,0,100,74),
26 DDBLTFAST_WAIT|DDBLTFAST_SRCOLORKEY);
27 }
```

**程序说明**

ShipMove()函数主要按照玩家移动鼠标的信息来重设飞机的贴图坐标 (ship.x,ship.y), 并设定飞机区域的范围为 ship.r[0]~ship.r[2], 最后第 25、26 行程序代码则按照新的坐标来贴上飞机图。

PasteMon()函数的主要功能是按照各种怪物类型来决定其贴图坐标, 然后再贴上怪物图。程序说明如下所示。

## 程序代码

```

1 void canvasFrame::PasteMon()
2 {
3     for(i=0;i<50;i++)
4         if(mon[i].exist)
5         {
6             switch(mon[i].type)
7             {
8                 case 0:
9                     if(mon[i].draw%10 == 0)
10                    {
11                        mon[i].draw = 0;
12                        mon[i].x += 20;           //往右移动
13                        if(mon[i].x >= 640)
14                        {
15                            mon[i].exist = false;
16                            break;
17                        }
18                        if(rand()%2 == 0)         //以随机数决定往上或往下移动
19                        {
20                            mon[i].y -= 20;
21                            if(mon[i].y<0)         //移动到最上方
22                                mon[i].y = 0;
23                        }
24                        else
25                        {
26                            mon[i].y +=20;
27                            if(mon[i].y>420)       //移动到最下方
28                                mon[i].y = 420;
29                        }
30                        if(mon[i].x <100)           //判断是否发射子弹
31                            for(j=0;j<100;j++)
32                            {
33                                if(!c[j].exist)
34                                {
35                                    c[j].x = mon[i].x + 61;
36                                    c[j].y = mon[i].y+30;
37                                    c[j].exist = true;
38                                    ccount++;
39                                    break;
40                                }
41                            }
42                        }
43                        mon[i].draw++;
44                        DDBuf->BlitFast(mon[i].x,mon[i].y,DDPla[3],CRect(0,0,80,59),
45                            DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);
46                        break;

```

```

47     case 1:
48         if(mon[i].draw%5 == 0)
49         {
50             mon[i].draw = 0;
51             if(mon[i].x > ship.x)
52                 mon[i].x-=30;
53             else
54                 mon[i].x+=30;
55             if(mon[i].y > ship.y)
56                 mon[i].y-=30;
57             else
58                 mon[i].y+=30;
59
60         }
        /*设定怪物区域与碰撞点的程序代码(略)*/
61     mon[i].draw++;
62     DDBuf->BltFast(mon[i].x,mon[i].y,DDPla[4],CRect(0,0,60,56),
63         DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);
64     break;
65     case 2:
66         if(mon[i].draw%10 == 0)
67         {
68             mon[i].draw = 0;
69             mon[i].x -= 20;           //向左移动
70             if(mon[i].x < 0)
71             {
72                 mon[i].exist = false;
73                 break;
74             }
75         }
        /*设定怪物区域与碰撞点的程序代码(略)*/
76     mon[i].draw++;
77     DDBuf->BltFast(mon[i].x,mon[i].y,DDPla[5],CRect(0,0,79,51),
78         DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);
79     break;
80 }
81 }
82 }

```

#### 程序说明

(1) 第 3 行中 for 循环会取出 mon 数组中的所有怪物元素,若第 4 行判断式中的条件式“mon[i].exist”成立,则第 6 行程序代码便依怪物的类型来计算其贴图坐标并进行贴图。

(2) 在 monster 结构中定义了一个 draw 资料成员用来控制怪物移动的贴图速度,如第 9、48、66 行程序代码便是判断条件式“mon[i].draw%除数==0”是否成立来决定是否重新计算贴图坐标。若除数的值越小,重新计算贴图坐标的频率就越高,该类型的怪物移动速度就越快。

(3) 先来看 case 0 蜗牛的移动状况,若“mon[i].draw%10==0”条件成立,则第 12 行程序代码重设怪物的 X 坐标为“mon[i].x+=20”,倘若新的 X 坐标大于 640,则表示怪物已经跑到窗口外,第 15 行程序代码便将 mon[i].exist 设为 false,表示怪物已不存在。

(4) 蜗牛在 Y 轴上的移动方向则是依随机数而定, 若第 18 行的条件式 “rand()%2==0” 成立, 则往上移动 (mon[i].y-=20), 否则便往下移动 (mon[i].y+=20)。

(5) 第 30 行判断蜗牛在 X 轴上的坐标小于 100 时, 接下来的第 31~41 行程序代码便会在怪物子弹的数组 c 中填入一个新的元素, 即为蜗牛所发射的子弹。

(6) 每一个 case 的最后, 便按照怪物的所在坐标贴上该类型怪物的图案, 如第 44、45 行程序代码便是贴上蜗牛的图案。

(7) 若 mon[i].type 等于 1, 即怪物为小鸟, 便运行 case 1 下的程序代码, 其中 51~58 行程序代码设定其贴图坐标越来越接近飞机。

(8) 若 mon[i].type 等于 2, 即怪物为鱼, 便运行 case 2 下的程序代码, 其中第 69 行程序代码设定鱼在 X 轴上的贴图坐标每次向左 20 个像素点, Y 轴方向上的坐标不变。

(9) 当运行完整个循环之后, 所有还存在的怪物便都会贴到绘图页中。

Shoot()函数主要的作用为判断玩家是否 Click 鼠标左键来发射子弹, 由于 DirectInput 只能检测玩家是否按下鼠标按键, 而无法检测是否是做 Click 鼠标按键的动作, 因此在这个函数中使用了一个布尔变量 “press” 来判断 Click。玩家在按下鼠标左键发射一颗子弹后必须松开, 下一次按下时才能发射另一颗子弹。下面看看程序说明。

## 程序代码

```
1 void canvasFrame::Shoot()
2 {
3     if(DIstate.rgbButtons[0] & 0x80)           //判断是否按下鼠标左键
4     {
5         if(!press)
6         {
7             for(i=0;i<100;i++)
8             {
9                 if(!b[i].exist)
10                {
11                    DSBuf[1]->Play(0,0,0);       //播放发射子弹的声音
12                    b[i].x = ship.x;
13                    b[i].y = ship.y+30;
14                    b[i].exist = true;
15                    bcount++;
16                    break;
17                }
18            }
19            press = true;
20        }
21    }
22    else
23        press = false;
24 }
```

## 程序说明

(1) 第 3 行: 判断玩家是否按下鼠标左键, 若条件式不成立, 则第 23 行程序代码将 press 设为 false, 如此下一次按下鼠标左键时便可发射子弹。

(2) 第 5 行: 判断 press 必须为 false 才会运行接下来的程序代码发射子弹。

(3) 第 7~18 行: for 循环会取出飞机子弹数组中的元素, 若某一元素中的子弹不存在, 则第 12~14 行程序代码在该元素中填入一个子弹, 第 15 行程序代码则累加目前飞机子弹的总数。

(4) 第 19 行: 在发射子弹后将 press 设为 true, 如此玩家必须松开鼠标左键让 press 设为 false, 下一次才能再发射子弹。

MonBombShip()函数判断怪物是否与飞机发生碰撞, 若目前在怪物数组中还存活的任一怪物与飞机发生碰撞, 游戏将会结束。程序内容说明如下所示。

#### 程序代码

```

1 void canvasFrame::MonBombShip()
2 {
3     if(mcount != 0)                //判断是否有怪物存在
4     for(i=0;i<50;i++)              //取出所有怪物
5     if(mon[i].exist)
6         for(j=0;j<=3;j++)          //取出所有怪物的碰撞点
7         for(m=0;m<3;m++)           //取出飞机各个区块
8             if(mon[i].p[j].x >= ship.r[m].left && mon[i].p[j].x
9             <= ship.r[m].right && mon[i].p[j].y >= ship.r[m].top
10             && mon[i].p[j].y <= ship.r[m].bottom)
11             {
12                 DDBuf->BlitFast(mon[i].p[j].x-40,mon[i].p[j].y-40,
13                 DDPla[7],CRect(0,0,80,80),DDBLTFAST_WAIT|
14                 DDBLTFAST_SRCCOLORKEY);
15                 DDBuf->BlitFast(0,150,DDPla[8],CRect(0,0,640,159),
16                 DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);
17                 DSBuf[0]->Stop();    //停止背景音乐
18                 DSBuf[3]->Play(0,0,0); //播放碰撞声音
19                 for(i=0;i<50;i++)    //清空怪物数组
20                     if(mon[i].exist)
21                         mon[i].exist = false;
22                 for(i=0;i<100;i++)    //清空怪物子弹数组
23                     if(c[i].exist)
24                         c[i].exist = false;
25                 for(i=0;i<100;i++)    //清空飞机子弹数组
26                     if(b[i].exist)
27                         b[i].exist = false;
28                 bcount = 0;
29                 ccount = 0;
30                 mcount = 0;
31                 over = true;          //游戏结束
32                 break;
33             }
34 }
```

#### 程序说明

(1) 第 4 行: for 循环取出怪物数组中的所有怪物。

(2) 第 5 行: 判断式以目前还存活的怪物来判断是否与飞机发生碰撞。

(3) 第 6 行: for 循环——取出怪物的各个碰撞点来判断是否在飞机区块的范围内。

(4) 第 8~10 行: 此判断式成立的话, 则碰撞点便在飞机的区块中, 表示发生碰撞。

(5) 第 12~14 行: 贴上爆炸画面。

(6) 第 15、16 行: 贴上游戏结束画面。

(7) 第 31 行: 将 over 设为 true, 表示游戏结束。

HitShip()函数用来判断怪物所发射的子弹是否打中飞机, 若目前在怪物子弹数组中还存在的任一颗子弹打中飞机, 游戏将会结束。程序内容说明如下。

## 程序代码

```

1 void canvasFrame::HitShip()
2 {
3     if(ccount !=0)
4         for(i=0;i<100;i++)                //取出所有怪物子弹
5             if(c[i].exist)
6                 {
7                     c[i].hitx = c[i].x + 10;
8                     c[i].hity = c[i].y + 5;
9                     hit = false;
10                    for(j=0;j<=2;j++)        //取出飞机各个区块
11                        {
12                            if(c[i].hitx >= ship.r[j].left && c[i].hitx <= ship.r[j].right
13                            && c[i].hity >= ship.r[j].top && c[i].hity <= ship.r[j].bottom)
14                                {
15                                    hit = true;
16                                    DDBuf->BltFast(ship.x,ship.y,DDPla[7],CRect(0,0,80,80),
17                                    DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);
18                                    break;
19                                }
20                        }
21                    if(hit)
22                        {
23                            DDBuf->BltFast(0,150,DDPla[8],CRect(0,0,640,159),
24                            DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);
25                            DSBuf[0]->Stop();        //停止背景音乐
26                            DSBuf[3]->Play(0,0,0);    //播放碰撞声音
27                            for(i=0;i<50;i++)        //清空怪物数组
28                                if(mon[i].exist)
29                                    mon[i].exist = false;
30                            for(i=0;i<100;i++)        //清空怪物子弹数组
31                                if(c[i].exist)
32                                    c[i].exist = false;
33                            for(i=0;i<100;i++)        //清空飞机子弹数组
34                                if(b[i].exist)
35                                    b[i].exist = false;
36                            over = true;                //游戏结束
37                            bcount = 0;
38                            ccount = 0;
39                            mcount = 0;
40                            break;

```

```

41         }
42     else
43     {
44         c[i].x +=5;           //计算下次 x 坐标
45         if(c[i].x>=640)       //子弹移动超出屏幕
46         {
47             c[i].exist = false;
48             ccount--;
49         }
50     }
51 }
52 }

```

### 程序说明

- (1) 第 4 行: for 循环取出怪物子弹数组中的所有子弹。
- (2) 第 5 行: 判断式以目前还存在的怪物子弹来判断是否打中飞机。
- (3) 第 7、8 行: 依子弹目前的位置计算出其碰撞点。
- (4) 第 10~20 行: 判断子弹的碰撞点是否在飞机的区块中。
- (5) 第 15 行: 将表示是否击中飞机的布尔变量 hit 设为 true。
- (6) 第 16、17 行: 贴上爆炸画面。
- (7) 第 21 行: 判断 hit 值是否为 true, 若 hit 值为 true, 则表示飞机被打中。
- (8) 第 23~40 行: 运行结束游戏的动作。
- (9) 第 44~49 行: 如果子弹并没有打中飞机, 则此程序代码计算该颗子弹下一次出现的 X 坐标, 如果计算后的 X 坐标大于等于 640, 则表示子弹移动已超出窗口, 便将子弹设为不存在 (c[i].exist=false)。

即 HitMon()函数用来判断飞机所发射的子弹是否打中怪物, 这个函数将会判断目前还存在的每一颗飞机子弹与还存活的每一只怪物是否发生碰撞, 并按照子弹所击中的怪物类型来累加游戏分数, 程序说明如下所示。

### 程序代码

```

1  void canvasFrame::HitMon()
2  {
3      if(bcount != 0)
4          for(i=0;i<100;i++)           //取出所有飞机子弹
5          {
6              if(b[i].exist)
7              {
8                  b[i].hitx = b[i].x + 3;
9                  b[i].hity = b[i].y + 5;
10                 hit = false;
11                 for(m=0;m<50;m++)       //取出所有怪物
12                 {
13                     if(mon[m].exist)
14                     {
15                         for(j=0;j<=2;j++)   //取出怪物各个区块
16                         {
17                             if(b[i].hitx >= mon[m].r[j].left && b[i].hitx <= mon[m].r[j].right

```



```

18         && b[i].hity >= mon[m].r[j].top && b[i].hity <= mon[m].r[j].bottom)
19     {
20         hit = true;
21         DDBuf->BlitFast(mon[m].x, mon[m].y, DDPla[7], CRect(0, 0, 80, 80),
22             DDBLTFAST_WAIT|DDBLTFAST_SRCCOLORKEY);
23         switch(mon[m].type)
24         {
25             case 0:           //打中蜗牛
26                 score+=100;
27                 break;
28             case 1:           //打中小鸟
29                 score+=50;
30                 break;
31             case 2:           //打中鱼
32                 score+=150;
33             }
34             break;
35         }
36     }
37     if(hit)
38     {
39         mon[m].exist = false;
40         b[i].exist = false;
41         DSBuf[2]->Play(0, 0, 0);
42         mcount--;
43         bcount--;
44         break;
45     }
46 }
47 }
48 if(hit)
49     continue;
50 b[i].x -=20;           //计算下次 x 坐标
51 if(b[i].x<-10)         //子弹移动超出屏幕
52 {
53     b[i].exist = false;
54     bcount--;
55 }
56 }
57 }
58 }

```

### 程序说明

- (1) 第 6 行：以目前还存在的飞机子弹来判断是否打中怪物。
- (2) 第 8、9 行：计算子弹的碰撞点。
- (3) 第 11~47 行：for 循环以目前还存活的怪物来判断是否被子弹击中。
- (4) 第 17、18 行：判断式成立时，表示子弹打中怪物，此时 hit 变量会被设为 true。
- (5) 第 23~33 行：switch 判断式判断被打中的怪物类型以累加总分“score”。

(6) 第 38~44 行: 设定怪物死亡 (`mon[m].exist=false`), 子弹消失 (`b[i].exist=false`), 并递减存活怪物与飞机子弹的总数。

`OnKeyDown` 函数的作用为当游戏结束时, 玩家可以按下【F1】键来重新开始游戏。下面来看看处理使用者按下按键信息 `OnKeyDown` 中函数中的内容。

#### 程序代码

```

1 void canvasFrame::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
2 {
3     if( nChar== VK_ESCAPE )           //按下【Esc】键
4         PostMessage(WM_CLOSE );       //结束程序
5     if( nChar == VK_F1 )               //按下【F1】键
6     {
7         over = false;
8         score = 0;
9         ship.x = 540;
10        ship.y = 200;
11        ship.r[0].left = ship.x + 16;
12        ship.r[0].top = ship.y + 3;
13        ship.r[0].right = ship.x + 50;
14        ship.r[0].bottom = ship.y + 53;
15        ship.r[1].left = ship.x + 8;
16        ship.r[1].top = ship.y + 16;
17        ship.r[1].right = ship.x + 78;
18        ship.r[1].bottom = ship.y + 50;
19        ship.r[2].left = ship.x + 35;
20        ship.r[2].top = ship.y + 35;
21        ship.r[2].right = ship.x + 81;
22        ship.r[2].bottom = ship.y + 71;
23        DSBuf[0]->Play(0,0,1);         //回放背景音乐
24    }
25    CFrameWnd::OnKeyDown(nChar, nRepCnt, nFlags);
26 }
```

#### 程序说明

(1) 第 3、4 行: 设定按下【Esc】键则结束程序。

(2) 第 5 行: 设定当玩家按下【F1】键时, 第 7 行程序代码会将 `over` 设为 `false`, 如此 `OnTimer` 函数中进行游戏的程序代码便会被运行。

(3) 第 8、9、10 行: 重设游戏总分 (`score`) 为 0, 重设飞机初始的贴图坐标 (`ship.x, ship.y`)。

(4) 第 11~22 行: 按照初始坐标 (`ship.x, ship.y`) 设定飞机的区域。

(5) 第 23 行: 播放背景音乐, 重新开始进行游戏。

在这个太空射击游戏的范例中, 包含了几个射击游戏制作的重点, 如碰撞检测、怪物的产生、怪物移动、怪物自动攻击等等, 可以说构成了一个完整射击游戏的框架。学会了这些技巧后, 相信只要再多花点时间与心思, 必定可以设计出更出色的射击游戏。



## 附录 B 专业词汇

- **台片**  
俗称台湾盗版货的意思。
- **魔法力**  
即 MP 或 Magic Point，俗称“魔法力”，即游戏人物的魔法数值。
- **必杀技**  
在格斗游戏中，利用特殊按键所使用的克敌必胜技巧。
- **超必杀技**  
在格斗游戏中比必杀技还来得强大的技术。
- **密技**  
在游戏中故意设定的一些不为人知的技巧。
- **生命力**  
即 Hit Point (HP)，表示人物或作战单位的生命数值。
- **关卡**  
即游戏中一个连续的完整舞台、场景。
- **出招**  
是指游戏进行中人物的出招或打斗技巧。
- **贱招**  
是指使用重复的招术让对手面临毫无反击的状况。
- **回合**  
即 Round，是表示在格斗类游戏中的一个较量回合。
- **大头目**  
或称 Boss，指游戏中所出现最为难缠，而且仅出现一次的对手。
- **第一人称视角**  
是以游戏主角的亲身经历所进行的角度，通常在游戏屏幕中不出现主角的身影，让玩家感觉自己就是游戏中的主角。
- **第二人称视角**  
或称追尾视角，指紧随游戏主角的背影角度来描述。
- **第三人称视角**  
是以一个旁观者的角度来描述或观看游戏的发展。
- **游戏攻略**  
或称 walkthrough，是指较完整的游戏攻击指南。
- **死路**  
指游戏在进行一定程度后，突然发现没有继续下去的线索。
- **游荡**  
玩家在游戏地图移动时，迟迟无法发现进入游戏下一步的途径。

- NPC

Non Player Character 的缩写，是指在一个游戏世界中，主角以外的特定人物，或称为非玩家角色。

- MUD

俗称多用户地牢，即全球信息网上多人参与的实战游戏。

- 剧情

即 storyline，游戏的故事大纲。

- 改机

改成“台片”可以适用的主机，也就是玩盗版游戏的主机。

- 金手指

是一种可以让游戏中的某些设定数值改变的外围道具。

- RPG

指角色扮演游戏。

- AVG

指冒险类游戏。

- SIM

指模拟类游戏。

- SFC

SFC 即为 Super Family Computer 的缩写，是 FC 的下一代，由任天堂公司生产发行的 16 位 TV 游戏机，中译为“超级家用计算机”，或俗称超任游戏主机。

- MD

MD 即为 MEGA Drive 的缩写，它是世嘉（SEGA）公司生产的 16 位 TV 游戏机，中译名是“兆位驱动”。

- SS

是 SEGA SATURN 的缩写，是世嘉公司生产的 32 位 TV 游戏机，中译为“世嘉土星”。

- PS

PS 即 Play Station 的缩写，是 Sony 公司生产的 32 位 TV 游戏机，中译为“玩家游戏站”。

PS 主机的历史，堪称电玩史上的一大奇迹。由于这款游戏主机的推出，使得风评一直都不错的 32 位 SS 游戏主机遭受到前所未有的打击，许多游戏的大作都移植到这一款游戏主机上。PS 游戏主机最大的特色就在于 3D 的运算方面，许多游戏都将 PS 游戏主机的 3D 性能发挥到极致，使得 PS 游戏主机在玩家的心中留下了非常深刻的印象。

- N64

N64 为 Nintendo 64 的缩写，是任天堂公司生产的 64 位 TV 游戏机，中译为“任天堂 64”。它也是第一台以四个操作接口为主体的 64 位运算游戏主机。

- DC

Dream Cast 的缩写，是由世嘉公司生产的 128 位游戏主机，缺点是很容易发热，经常有死机的状况。

- PS2

PS2 即为 Play Station 2 的缩写，它是由 Sony 公司生产发行的 128 位 PS 第二代主机。PS2 游戏主机与前面所提的游戏主机有很大不同，因为 PS2 游戏主机不但可以玩 PS 游戏主机上的所有游戏作品，更能够让玩家享受到视听娱乐等附加的功能，而且所有 PS 游戏主机上的所有

配备（手柄、记忆卡等）都可以继续使用，使得拥有 PS 游戏主机的玩家们还想再拥有 PS2 游戏主机。

- **NGC**

NGC 即为 Nintendo Game Cube 的缩写，是任天堂公司所生产的 128 位 TV 游戏机，中译为“任天堂游戏立方体”，低廉的售价是其最吸引玩家的地方。

- **GG**

即为 Game Gear 的缩写，是世嘉公司所生产的 8 位掌上型游戏机。

- **GB**

即为 Game Boy 的缩写，是任天堂 8 位掌上型游戏机，中译为“游戏小子”。

- **XBOX**

XBOX 是微软（Microsoft）公司生产发行的 128 位 TV 游戏机。

XBOX 也是微软公司本世纪推出的新的游戏平台系统，它带给玩家们有史以来最具震撼力的游戏体验。XBOX 不但使游戏设计高手的功力更加强大，带来前所未有的创新技术，进而创造出幻觉与现实界线变得模糊的游戏。

NGC、PS2 与 XBOX 是目前市面上最新、最流行的电玩游戏主机。它们彼此之间互相竞争，都想要在电玩领域上争得冠军的宝座。不论在硬件上、或者价格上都拼得你死我活，形成了现代游戏主机市场的肉搏战。

- **GBA**

GBA 即为 Game Boy Advance 的缩写，是由任天堂公司推出的新一代便携式掌上游戏机，具备了许多不同颜色的主机供玩家们选购。而且主机的外型也和历代的 GB 颇有分别，最大的分别是屏幕的位置改放在十字键和【A】、【B】键之间。

- **OpenGL**

OpenGL 即是 Open Graphics Libraries 的缩写，是一套计算机三维图形处理函数库。由于它是由各家显示厂商所共同定义的通用函数库，所以也是绘图成像的工业标准。

OpenGL 是由 SGI 公司为图形工作站开发的“IRIS GL”在跨平台移植的过程中发展而成的。SGI 公司在 1992 年 7 月的时候发布了 OpenGL 1.0 版之后，“OpenGL”即成为显示工业的一项独立标准。接下来“OpenGL”由成立于 1992 年的独立财团 OpenGL Architecture Review Board（ARB）控管。SGI 与 ARB 成员以投票方式产生这项独立标准，并制作成规范文件档（Specification）公诸于众，从此各家软硬件厂商便依据这种原则标准来开发自己系统上的显示功能。各家软硬件厂商的产品只要通过了 ARB 的全部规范标准测试之后，产品才能称为支持 OpenGL 的成像技术。

- **DirectX**

DirectX 是一种 Windows 系统的应用程序接口（Application Programming Interface，简称 API），可以让以 Windows 为平台的游戏或多媒体程序获得更高的运行效率，而且还可以加强 3D 图形成像和声音效果。另外，它提供给设计人员一个共同的硬件驱动标准，让游戏开发者不必为每一个厂家的硬件设备编写不同的驱动程序，同时也降低了使用者安装及设置硬件的复杂度。

- **MIDI**

MIDI 又可以称为“电子乐器数字信号”，由“MIDI 生产商协会”（MIDI Manufacturers Association）制订。其制订的数据是为了给所有 MIDI 仪器制造商提供音色与打击乐器音效的清单标准。MIDI 共有 128 个标准音和 81 个打击乐器的音色。

- DSP

DSP 即 Digital Signal Processing 的缩写, 中译为“数字信号处理”。DSP 是声卡中专门用来处理效果的芯片, 又可以称之为“效果器”。由于具有这种芯片的声卡价格比较昂贵, 所以通常只有在较高级的声卡中才会看到。

- 波表合成

波表合成会通过乐器的声音进行取样, 并将取样的数据保存下来。播放音效时就靠声卡上的微处理器或者 CPU 来处理这些数据的发声。

- SNR

SNR 即为 Signal to Noise Ratio 的缩写, 中译为“信噪比”。它是一个诊断声卡抑制噪音能力的重要指标。SNR 是有用信号和噪声信号功率的比值, 其单位是分贝。SNR 的值越大, 则声卡的滤波效果越好, 所以优秀声卡的 SNR 值至少要大于 80 分贝。

- FM 合成

FM 合成技术是早期电子合成乐器所采用的发音方式, 后来由 Yamaha 公司将其应用到 PC 声卡上。FM 比 PC 喇叭所提供的效果要好, 其最大的特色就是 FM 的发音方式使声音听起来比较干净、清脆。

- 5.1 声道

5.1 声道已经被广泛地运用在各种电影院及家庭中, 一些比较知名的声音录制压缩格式, 例如“杜比 AC-3” (Dolby Digital)、DTS 等都是以 5.1 声道系统为技术的蓝本。其实 5.1 声道系统的构思来自于 4.1 环绕系统, 两者不同之处在于 5.1 声道增加了一个中置单元。这个中置单元是负责传送低于 80Hz 的声音信号, 在影片播放的时候更有利于加强人声, 其原理就是将人的对话集中在整个环境的中央以提高整体效果。

- A3D

A3D 是由 Aureal Semiconductor 所开发的一套交互式 3D 定位音效技术, 它的主要特色是在 3D 音效定位上有非常独特的处理。最早的 A3D 技术只需要两个音箱为输出所用, 不过在新的版本中, 如果要想实现 A3D 的效果时就必须使用四个音箱才能达成。

- DAC

DAC 即 Digital-analog Converter 的缩写, 中译为“数字仿真转换器”。因为一般的音响都只能接受模拟信号的数据, 而计算机中所处理的数据通常是数字信号, 因此声卡在读出数字信号后, 必须通过 DAC 的转换使其成为一般音响能够接受的模拟信号, 再由音响带动音箱发出声音。

- EAX

EAX 即 Environmental Audio Extensions 的缩写, 中译为“环境音效”。它是由创新与微软这两家公司所联合提出的作为 DirectSound 3D 扩展的一套应用程序技术。EAX 最新的 2.0 版可以实现混音、封闭、阻塞等效果。混音呈现让虚拟音源随着环境的变化而产生不同的效果; 封闭可以模拟听者与音源之间有大面积阻碍的效果; 阻塞是模拟听者与音源之间有小面积阻隔的效果。不过, 如果要使用这一技术的话, 必须具备 4 个音箱才能表现出 EAX 的特殊效果。

- FR

FR 是 Frequency Response 的缩写, 中译为“频率响应”。FR 是声卡 DAC 转换器频率响应能力的评价指标。

- CS

CS 即是 Counter-Strike 的缩写, 中译为“反恐精英”。CS 最大的特点就在于它对现实的

真实模拟。由于武器和游戏规则都与现实中的情景接近，随之而来也发展出各式各样的战术，例如“突击”、“偷袭”、“蹲射”、“跳射”、“左右闪”、“AK 狙击”及“沙漠狙击”等，这些技巧难以胜数。当然，还可以在 CS 的游戏中进行小队的战术技巧，例如“交叉掩护”或“分进合击”等常见的战术。毕竟 CS 是一个虚拟的游戏，而在现实生活中，如果没有一个好的团队，那是不可能造就出今日举世闻名的世界级游戏大作的。

- **倒叙法**

倒叙法就是将玩家先处于事件发生后的结果之中，然后再让玩家回到过去去了解事件发生的原因，或者让玩家自行去阻止事件发生。如“MYST”（迷雾之岛）这一 AVG 游戏就是最典型的例子。

- **正叙法**

正叙法就是以平铺直叙的表达方式，让游戏故事情节随玩家们的遭遇而展开。也就是说，玩家对于游戏进行中的一切都是未知的，而未来的发展只等待玩家自己去发现或创造。通常在游戏中会看到它们多半以这样的陈述方式来描述游戏故事剧情的。

- **DirectX SDK**

Microsoft DirectX 提供了一套非常好用的应用程序接口，其中包含了设计高性能、实时应用的程序代码，它就是“DirectX SDK”（DirectX Software Development Kit，俗称“DirectX 开发包”）。DirectX SDK 技术能够帮助我们轻易地建构计算机游戏和多媒体的应用程序，其中包括 DirectDraw、DirectSound、DirectPlay、Direct3D 和 DirectInput 等部分的 API 命令及媒体相关的组件（Component）。

- **Direct3D**

Direct3D 简称 D3D，对于现在的游戏来说，D3D 实在太重要了。由于 3D 游戏的兴起，各大厂商纷纷推出 3D 显示加速卡，而为了让显示卡能够避免如同声卡规格的统一性，微软从 DirectX3.0 以后就加入 D3D 这个 API 的技术，让 3D 游戏有一个共同的开发标准。如此一来，当游戏在运行的时候，如果需要使用到绘图的部分时，DirectX 就会通过 D3D 向显示卡驱动程序提出成像要求，并且让显示卡完成绘图的动作。

- **DirectDraw**

DirectDraw 是 DirectX 中非常重要的一部分。它担任的工作是 2D 图形的处理。在以前 DOS 环境下设计游戏的时候，考虑到游戏的整体运行速度，让游戏程序直接对应硬设备的内存区段来加快运行速度。不过在 Windows 操作系统这种保护模式之下，所有图形的接口动作都必须经过 GDI 这个图形处理中心来处理，而不能直接对硬设备下命令。但是 GDI 接口对连续画面处理的效果非常差，如果说一款游戏经由 GDI 接口来处理，那么其成像的效果画面将会大打折扣。

- **GDI**

图形设备接口（Graphics Device Interface）负责在屏幕上显示图形所用的接口。GDI 是由百余个函数所组成的。GDI 函数必须通过设备描述表（简称 DC）的句柄来控制绘图。

- **MMX**

MMX 处理器是由 Intel 公司研发出来的 CPU 命令集，它含有专门处理声音、影像和图片的额外命令。

- **3DNow!**

AMD 公司利用一种更先进的 3D 算法提供主流 PC 上的 3D 应用服务，称之为“3DNow!”技术（x86 构架上的 3D 技术的创新）。简单地说，“3DNow!”技术是 x86 构架上的的一种创新，它突破 x86 架构的瓶颈，使计算机能够产生更加真实、有趣和生动的 3D 画面。



- AGP

计算机内部的图形卡专用插槽。它由 Intel 设计，传输速度比一般 PCI 卡快两倍以上。AGP 可以让特定的显示卡使用专用内存来存储或获取游戏、3D 应用程序的图片类型。

- 着色引擎

着色引擎 (shader) 是用着色语言编写的一段程序代码，着色语言是专门为有效顶点或图形组件中使用而设计的。

- DirectMusic

DirectMusic 的核心包含了“MIDI 播放系统”及“互动音乐系统”两个部分。当 Windows 加入了 DirectMusic 的功能之后，它对 MIDI 的支持能力便大大地提高了。因为以前 Windows 经常被频率及时序标签等问题所困扰，现在这些问题都迎刃而解了。并且 DirectMusic 也采用了最近正式通过的 DLS (Downloadable Sounds) 1.0 技术，所以也解决了 MIDI 音乐播放时效果不一致的老问题。另外，每首 MIDI 音乐可以使用任何自定的音源，并突破了 128 种 General Midi 的音源限制。

- DirectSound

DirectSound 是用于处理声音的 API 命令函数。除播放声音和处理混音外，加强了 3D 音效的部分，并且提供录音的功能。对于前面所提的声卡兼容的问题，现在可以利用 DirectSound 的技术来解决。

- DirectInput

DirectInput 用来处理游戏的一些外围设备装置，例如游戏杆、GamePad 接口、方向盘、VR 手套、力回馈等。以往要在 DOS 环境下使用方向盘玩赛车游戏的话，必须先调整好方向盘设备的 IRQ、DMA 等设定才能使用方向盘的装置，而现在 DirectInput 则可以提高这些设备与游戏配合的兼容性，不需要对外围装置做任何特别的设定。

- DirectPlay

DirectPlay 是为了满足目前流行的网络游戏而开发的 API 命令，而且它还支持许多通讯协议，让玩家可以利用各种联网的方式来进行网络游戏的对战。此外，DirectPlay 也提供网络对谈的功能，并包含保密的措施。

- TRPG

纸上角色扮演游戏 (Table talk - Role Playing Game, TRPG) 由纸上战略游戏演变而来。玩这一类的游戏时需要几个玩家来搭配，而在这几个玩家中必须选择一个主持人并准备一些纸片道具。游戏进行的时候，游戏者需要以掷骰子来决定前进的步数，再由主持人来讲述此游戏的故事内容，然后让玩家们知道他遇上什么事件。游戏中主持人就是游戏的灵魂，也是这个游戏的创作者与故事讲述者，同时也是规则的解释人。所有的玩家等于是扮演主持人故事中的一个特定角色，而这个故事的精彩与否则直接取决于主持人的能力。以投掷骰子的方式，体验不可预知的结果和玩家的行动，这是角色扮演游戏最原始的雏形。

- ACT

动作类游戏又称为“ACT” (Action Game)，在游戏界占有最大市场的游戏类型。只要是游戏玩家，相信一定不会排斥这种动作类游戏的玩法，因为动作类游戏是所有游戏类别最基本的游戏玩法模式。

- ARPG

“动作角色扮演” (Action Role Playing Games, ARPG) 发展的时间较 RPG 游戏与动作游戏更晚，因为它采取动作游戏紧凑的玩法与 RPG 游戏剧情的流程为主线，让玩家可以体会动

作游戏的刺激感与 RPG 游戏的角色扮演的乐趣，所以它让游戏产业再度掀起一股独特的风潮。

- **STA**

策略类游戏 (Strategy Game, STA) 也是属于让玩家将思考融于娱乐中的一种游戏类别。早期的策略型游戏以战棋为主，如象棋、军棋等。主要是让玩家在一场特定的地形之中，运用自己的思路来布置属于自己的棋子以打败对方为目的。早期的战棋游戏是对方走一步，玩家才能走一步，而如今的策略型游戏加入了实时的游戏机制。简单的说，在一个特定的地形中，玩家在做内政、军事的整备，对方也一样可以做相同动作的整备，不会因为玩家没有动作，对方就不动作，相当符合如今的战略游戏趋势。

- **VRML**

虚拟现实建模语言 (Virtual Reality Modeling Language, VRML)，是一种 Web 上描述三维空间的标准表示方法。VRML 使得人们可以利用 HTML 的技术来为 3D 空间建模，然后由浏览器的实际计算来表现这些模型。VRML 并不是一种命令语言，而是某种图像文件格式。

- **粒子系统**

所谓的粒子系统是将大自然中的物体运动和自然现象用一系列的粒子来描述，再将这些粒子运动的轨迹映现到显示屏幕上，接着便可以在屏幕上看到物体运动和自然现象的模拟效果了。

粒子系统可以在屏幕上表现出许多特殊效果，例如火焰、火苗、瀑布、雪花飞舞等。其实粒子系统不怕做不到，只怕想不到而已，只要发挥丰富的想象力便可以创造出意想不到的效果。

- **地图编辑器**

在一套游戏的制作过程当中，不管是要开发 2D 或是 3D 的游戏，都需要使用地图编辑器来编制游戏中的场景。地图编辑器是策划人员将游戏中所需的场景元素告诉程序设计者与美工人员，然后程序设计者利用美工人员绘制出来的图素，再编写一个可以编辑此套游戏场景的应用程序，而这个应用程序即可在策划人员编制游戏场景时使用。

- **粒子编辑器**

粒子编辑器用来编辑游戏中的所有粒子特效。通过它程序设计者就不必再大费周章地编写一大堆程序代码来发展游戏的炫目效果。



## 附录 C 常用 Windows 虚拟键表

虚拟键码	对应按键
VK_LBUTTON	鼠标左键
VK_RBUTTON	鼠标右键
VK_MBUTTON	鼠标中键
VK_BACK	Backspace
VK_TAB	Tab
VK_RETURN	Enter
VK_SHIFT	Shift (不分左右)
VK_CONTROL	Ctrl (不分左右)
VK_PAUSE	Pause、Break
VK_ESCAPE	Esc
VK_SPACE	空格键
VK_PRIOR	Page Up
VK_NEXT	Page Down
VK_END	End
VK_HOME	Home
VK_LEFT	←
VK_UP	↑
VK_RIGHT	→
VK_DOWN	↓
VK_INSERT	Insert
VK_DELETE	Delete
VK_LWIN	左 Windows 键
VK_RWIN	右 Windows 键
VK_APPS	应用程序键
VK_NUMPAD0	数字键 0
VK_NUMPAD1	数字键 1
VK_NUMPAD2	数字键 2
VK_NUMPAD3	数字键 3
VK_NUMPAD4	数字键 4
VK_NUMPAD5	数字键 5
VK_NUMPAD6	数字键 6
VK_NUMPAD7	数字键 7
VK_NUMPAD8	数字键 8
VK_NUMPAD9	数字键 9

虚拟键码	对应按键
VK_MULTIPLY	数字键*
VK_ADD	数字键+
VK_SUBTRACT	数字键-
VK_DECIMAL	小数点.
VK_DIVIDE	数字键/
VK_F1	F1
VK_F2	F2
VK_F3	F3
VK_F4	F4
VK_F5	F5
VK_F6	F6
VK_F7	F7
VK_F8	F8
VK_F9	F9
VK_F10	F10
VK_F11	F11
VK_F12	F12
VK_NUMLOCK	Num Lock
VK_SCROLL	Scroll Lock
VK_0~VK_9	0~9
VK_A~VK_Z	A~Z